

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка додатку для аналізу структури файлової системи та візуалізації використання дискового простору

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших
авторів без відповідних
посилань.

Студент гр. ІПЗ–22–1

_____ /М. О. Таточенко/

Керівник кваліфікаційної
роботи

_____ / Н. О. Карабут /

В.о. завідувача кафедри

_____ / А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ–22–1 Таточенку Максиму Олександровичу

1. Тема: «Розробка додатку для аналізу структури файлової системи та візуалізації використання дискового простору» затверджена наказом по КНУ № 284с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «05» червня 2026 р.
3. Вихідні дані по роботі: створене програмне забезпечення має забезпечити візуалізацію використання простору постійного запам'ятовуючого пристрою та реалізувати відображення дерева директорій.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих на ринку аналогічних програмних продуктів, обґрунтувати функціонал розроблюваної системи, створити програмне забезпечення, здійснити тестування розробленого додатку.
5. Перелік ілюстративного матеріалу: функціональна схема, блок–схема алгоритму, зображення екранних форм додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	05.01.26 – 24.01.26
2	Аналіз існуючих методів вирішення проблеми	25.01.26 – 07.02.26
3	Формулювання актуальності роботи і постановка завдань	08.02.26 – 18.02.26
4	Оформлення матеріалів першого розділу роботи	18.02.26 – 02.03.26
5	Створення функціональної системи та алгоритму додатку	03.03.26 – 15.03.26
6	Оформлення матеріалів другого розділу роботи	16.04.26 – 09.04.26
7	Розробка баз даних, інтерфейсу програмного забезпечення, програмних модулів	10.04.26 – 01.05.26
8	Оформлення додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 04.06.26

Дата видачі завдання: «04» січня 2026 р.

Студент: _____ / М. О. Таточенко /

Керівник роботи: _____ / Н. О. Карабут /

РЕФЕРАТ

АНАЛІЗ ФАЙЛОВОЇ СИСТЕМИ, ДУБЛІКАТИ ФАЙЛІВ,
КРИПТОГРАФІЧНЕ ХЕШУВАННЯ, СТРУКТУРА КАТАЛОГІВ,
ВІЗУАЛІЗАЦІЯ ДАНИХ.

Пояснювальна записка: 63 с., 11 рис., 1 дод., 22 джерела.

Об'єкт дослідження – процес аналізу файлової системи та оптимізації використання дискового простору.

Предмет дослідження – алгоритми, методи та програмні інструменти для пошуку надлишкових даних, виявлення дублікатів та візуалізації розподілу пам'яті.

У кваліфікаційній роботі спроектовано та реалізовано програмний засіб мовою C# на базі платформи .NET. Розроблено оптимізований алгоритм потокового сканування каталогів із застосуванням механізму м'якої деградації для безперебійної обробки системних помилок доступу. Реалізовано ефективну двоступеневу систему виявлення істинних дублікатів файлів та динамічного пошуку найбільших об'єктів. Розроблено графічний інтерфейс, який інтегрує ієрархічні дерева каталогів, таблиці та інтерактивні кругові діаграми для візуалізації зібраної статистики.

Практичне значення одержаних результатів полягає в тому, що розроблена програма є інструментом, який може бути використаний системними адміністраторами та звичайними користувачами для регулярного моніторингу стану жорстких дисків та виявлення критичних витрат пам'яті.

ABSTRACT

FILE SYSTEM ANALYSIS, FILE DUPLICATES, CRYPTOGRAPHIC HASHING, DIRECTORY STRUCTURE, DATA VISUALIZATION.

Explanatory note: 63 p., 11 fig., 1 app., 22 sources.

Object of research – the process of file system analysis and disk space utilization optimization.

Subject of research – algorithms, methods, and software tools for locating redundant data, detecting duplicates, and visualizing storage distribution.

In the qualification work, a software tool was designed and implemented in C# based on the .NET platform. An optimized algorithm for stream-based directory scanning was developed, utilizing a graceful degradation mechanism for the uninterrupted handling of system access errors. An effective two-stage system for detecting true file duplicates and dynamically searching for the largest objects was implemented. A graphical user interface was developed, integrating hierarchical directory trees, tables, and interactive pie charts to visualize the collected statistics.

The practical significance of the obtained results lies in the fact that the developed program is a tool that can be used by system administrators and regular users for the routine monitoring of hard drive status and the identification of critical storage consumption.

ЗМІСТ

ВСТУП	8
1 АНАЛІЗ МЕТОДІВ ДОСЛІДЖЕННЯ ФАЙЛОВИХ СИСТЕМ ТА ЗАСОБІВ ВІЗУАЛІЗАЦІЇ ДИСКОВОГО ПРОСТОРУ	10
1.1. Поняття та призначення файлової системи	10
1.2. Особливості сучасних файлових систем.....	11
1.3. Методи аналізу структури файлової системи.....	13
1.4. Методи пошуку дублікатів файлів	14
1.5. Методи візуалізації використання дискового простору.....	16
1.6. Аналіз існуючих програмних засобів та короткі висновки	17
2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ АНАЛІЗУ ФАЙлової СИСТЕМИ	20
2.1 Постановка задачі дослідження та формування вимог	20
2.2 Проєктування архітектури програмного засобу	21
2.3 Проєктування структури даних	24
2.4 Проєктування бази даних.....	26
2.5 Алгоритм роботи програми	28
3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ АНАЛІЗУ ФАЙЛІВ... 31	31
3.1 Обґрунтування вибору засобів розробки	31
3.2 Реалізація функціональних можливостей програмного засобу	32
3.2.1 Реалізація сканування та аналізу даних	32
3.2.2 Програмна реалізація пошуку дублікатів	33
3.2.3 Взаємодія з базою даних SQLite.....	33
3.2.4 Побудова інтерфейсу та візуалізація.....	34
3.3 Огляд роботи програми.....	34
ВИСНОВКИ	40

ПЕРЕЛІК ПОСИЛАНЬ	41
Додаток А - Програмний код	43

ВСТУП

Збільшення обсягів інформації створює певні складнощі для користувачів персональних комп'ютерів. Незважаючи на те, що ємність накопичувачів постійно зростає, проблема ефективного управління дисковим простором залишається актуальною. Особливо гостро це питання постає при використанні твердотільних накопичувачів, які забезпечують високу швидкість роботи операційної системи та програм, але з економічних причин часто мають значно менший обсяг пам'яті порівняно з класичними жорсткими дисками [1].

У процесі щоденної взаємодії з операційною системою накопичується велика кількість об'ємних медіафайлів, забутих архівів, тимчасових даних та повних копій існуючих документів. Залежно від специфіки роботи користувача, частка надлишкових або просто забутих даних може сягати значних масштабів та займати левову частку диска. Ручний пошук великих файлів та їхніх дублікатів, розпорошених по багаторівневих ієрархіях системних і користувацьких папок, є вкрай неефективним і забирає багато часу [2]. Це зумовлює потребу в автоматизації процесу очищення та аналізу за допомогою спеціалізованого програмного забезпечення.

Існуючі програмні рішення для моніторингу файлової системи часто мають перевантажений і незрозумілий для користувача інтерфейс або ж, навпаки, не надають достатньої гнучкості для глибокого структурного аналізу директорій та безпечної роботи з даними. Відповідно, розробка зручного додатку, здатного сканувати накопичувачі, візуалізувати структуру каталогів та автоматично виявляти надлишкові дані, є практично затребуваним завданням.

Мета дослідження полягає у створенні програмного рішення для комплексного аналізу файлової системи, яке дозволить користувачам швидко отримувати візуалізовану інформацію про розподіл дискового простору, виявляти найбільш об'ємні файли та знаходити дублікати даних.

Для досягнення поставленої мети було сформульовано такі завдання дослідження:

а) проаналізувати принципи організації сучасних файлових систем та методи ефективного обходу ієрархічних структур даних;

б) спроектувати багат шарову архітектуру додатку, яка забезпечить розділення бізнес-логіки, доступу до локальної бази даних та користувацького інтерфейсу;

в) розробити і програмно реалізувати алгоритми асинхронного сканування каталогів;

г) реалізувати пошук дублікатів на основі алгоритмів криптографічного хешування;

д) розробити модулі обробки статистики для візуалізації використання пам'яті за типами файлів та ієрархічної навігації;

е) створити зрозумілий графічний інтерфейс для зручної роботи зі знайденою інформацією.

Об'єктом дослідження є процес аналізу структури файлової системи та управління зберіганням цифрових даних.

Предметом дослідження виступають методи алгоритмічної обробки даних, підходи до порівняння файлів на основі хешування та програмні засоби для графічної візуалізації використання дискового простору.

Практичне значення результатів роботи полягає в тому, що зпроектований додаток зможе використовуватися для оптимізації зберігання інформації на персональних комп'ютерах. Застосування асинхронних підходів до програмування та оптимізованих алгоритмів читання директорій дозволить скоротити час сканування та знизити навантаження на систему під час пошуку дублікатів, а локальне збереження результатів аналізу дасть змогу швидко відстежувати зміни в дисковому просторі.

1 АНАЛІЗ МЕТОДІВ ДОСЛІДЖЕННЯ ФАЙЛОВИХ СИСТЕМ ТА ЗАСОБІВ ВІЗУАЛІЗАЦІЇ ДИСКОВОГО ПРОСТОРУ

1.1. Поняття та призначення файлової системи

Функціонування будь-якого комп'ютера неможливе без надійного механізму збереження інформації на енергонезалежних носіях. На фізичному рівні накопичувач є лише масивом секторів, що містять неструктуровані послідовності байтів [3]. Для перетворення цього масиву на зручне середовище для роботи з даними використовується файлова система – спеціалізована підсистема операційної системи, яка відповідає за організацію, зберігання та доступ до інформації [4]. Її головна роль полягає у наданні абстракції файлу як логічної одиниці даних, що приховує від користувача та прикладних програм складні апаратні деталі взаємодії з дисковим контролером.

Завдяки файловій системі операційна система може ефективно розподіляти вільний простір, відстежувати розташування фрагментів даних та забезпечувати розмежування прав доступу [5].

Для зручної навігації та впорядкування великих масивів інформації файлові системи використовують ієрархічну модель каталогів. Структура організації завжди починається з кореневого каталогу, який може містити файли та вкладені підкаталоги, утворюючи багатовимірне дерево [4]. Кожен об'єкт у цій ієрархії має унікальний шлях, що дозволяє точно ідентифікувати його розташування. Розуміння цієї деревоподібної структури є важливим етапом у розробленні програмного рішення. Щоб проаналізувати реальний вміст диска та візуалізувати його у вигляді інтерактивного дерева, система аналізу повинна мати механізми рекурсивного обходу всіх гілок цієї ієрархії, збираючи дані крок за кроком на кожному рівні вкладеності.

Окрім безпосередньо вмісту файлів, принципово важливим елементом збереження даних є метадані – службова інформація про характеристики збережених об'єктів. Метадані включають точний розмір файлу, його

розширення, дати створення та останньої модифікації, а також атрибути безпеки. Файлова система керує цими даними за допомогою спеціальних внутрішніх структур, таких як таблиці розміщення файлів або спеціалізовані індексні записи. Під час проєктування алгоритмів аналізу дискового простору необхідно спиратися саме на цю особливість організації даних. Зчитування виключно системних метаданих, а не фізичного вмісту кожного документа, дозволяє розроблюваному додатку формувати списки найважчих об'єктів та обчислювати статистику за типами даних, уникаючи надмірного навантаження на накопичувач.

1.2. Особливості сучасних файлових систем

Коректна робота програмного забезпечення для аналізу дискового простору безпосередньо залежить від розуміння архітектурних особливостей файлових систем, на яких розміщені дані користувача. Оскільки розроблюваний додаток функціонує в середовищі операційних систем сімейства Windows, ключову увагу необхідно приділити стандартам FAT32, exFAT та NTFS. Ці системи мають принципові відмінності у способах організації інформації, обмеженнях щодо розміру об'єктів і підходах до безпеки [6].

Файлова система FAT32 є однією з найстаріших серед тих, що досі активно використовуються. Її головна перевага полягає в абсолютній універсальності: накопичувачі з таким форматуванням підтримуються практично всіма наявними операційними системами та апаратними пристроями [7]. Проте архітектура FAT32 накладає жорстке обмеження на максимальний розмір одного файлу – він не може перевищувати 4 гігабайти. Це обмеження важливо враховувати під час тестування функціоналу пошуку найбільших файлів, оскільки на таких розділах апріорі неможливо знайти надвеликі медіафайли чи образи дисків, які часто стають причиною нестачі вільного місця.

Для подолання недоліків FAT32 було розроблено стандарт exFAT, оптимізований спеціально для портативних флеш-накопичувачів. Він усуває критичне обмеження на розмір файлу та значно підвищує ефективність розподілу вільного простору на зовнішніх пристроях [8]. Під час сканування портативних дисків програма-аналізатор стикається з відносно простою структурою метаданих exFAT, що дозволяє алгоритмам обходу каталогів працювати з максимальною швидкістю завдяки відсутності складної системи перевірки прав доступу.

Водночас стандартом де-факто для внутрішніх накопичувачів у середовищі Windows є NTFS. Ця файлова система використовує складну структуру на основі головної файлової таблиці Master File Table, підтримує журналювання для відновлення після збоїв і забезпечує деталізоване управління правами доступу на рівні окремих користувачів і процесів [6]. Саме наявність жорсткого розмежування прав доступу в NTFS зумовлює необхідність специфічної програмної обробки під час рекурсивного обходу каталогів. Під час сканування системних директорій програмне рішення повинно коректно перехоплювати помилки доступу (зокрема, винятки ізольованих середовищ), пропускаючи закриті системні папки без переривання загального потоку аналізу накопичувача.

Порівнюючи ці системи, можна виокремити чіткий розподіл сфер їхнього застосування. FAT32 та exFAT орієнтовані на максимальну сумісність і роботу зі знімними носіями, тоді як NTFS забезпечує надійність, безпеку та швидкодію для масивних внутрішніх дисків великого обсягу. Взаємодія з усіма цими стандартами на рівні розроблюваної системи здійснюється уніфіковано завдяки абстракціям платформи .NET. Використання базових класів роботи з вводом-виводом дозволяє програмі автоматично отримувати інформацію про логічні диски, їхній загальний обсяг та вільне місце незалежно від того, яка саме базова файлова система використовується на конкретному диску.

1.3. Методи аналізу структури файлової системи

Оскільки файлова система має яскраво виражену ієрархічну структуру, для її аналізу застосовуються класичні алгоритми обходу дерев і графів [9]. Найпоширенішим методом дослідження таких структур є рекурсивний обхід каталогів у глибину. Суть підходу полягає в тому, що алгоритм заходить у початкову директорію, знаходить перший вкладений підкаталог і відразу переходить до його обробки, повторюючи цей процес до досягнення кінцевого вузла, який не містить інших папок [10].

Під час реалізації цього методу на практиці виникає серйозна проблема: базові системні функції, які повертають одразу весь масив файлів у каталозі, споживають занадто багато оперативної пам'яті. Крім того, натрапивши на системну директорію з обмеженими правами доступу, такий алгоритм генерує критичний виняток, що призводить до аварійної зупинки всього процесу сканування. Тому розроблюване програмне рішення використовує оптимізовані потокові перелічувачі, які зчитують елементи по черзі. Це дозволяє програмно перехоплювати помилки доступу до ізолюваних областей і коректно пропускати їх, не перериваючи загальний хід аналізу накопичувача. Для збереження чуйності графічного інтерфейсу весь рекурсивний спуск виконується в окремих асинхронних потоках [11].

Після або під час отримання даних виникає завдання побудови дерева директорій для візуального представлення користувачеві. Спроба одночасно завантажити всі проскановані вузли в графічний компонент призведе до зависання вікна програми через величезну кількість об'єктів. Вирішенням є застосування патерна ледачого завантаження. Графічний інтерфейс системи аналізу відображає лише кореневі директорії. Інформація про вкладені підкаталоги прив'язується до візуального вузла виключно в момент, коли користувач натискає на нього для розгортання. Такий підхід гарантує миттєву реакцію інтерфейсу незалежно від глибини вкладеності реальної файлової системи.

Ще одним аспектом аналізу є обчислення розмірів каталогів. На рівні метаданих операційної системи папка не має власного розміру, вона є лише контейнером. Щоб дізнатися обсяг директорії, необхідно додати розміри всіх файлів, які лежать безпосередньо в ній, до суми розмірів усіх її підкаталогів. У розробленому алгоритмі це обчислення відбувається за принципом «знизу вгору» під час повернення з рекурсії: щойно алгоритм завершує сканування найглибшої папки, він передає її загальний розмір на рівень вище.

Окрім розмірів, процес сканування передбачає постійне визначення статистичних характеристик. Під час обходу кожного каталогу додаток групує файли за їхніми розширеннями. Це дозволяє сформувати частотний масив для подальшої візуалізації розподілу типів файлів на кругових діаграмах. Паралельно з цим ведеться пошук найбільших об'єктів. Замість того, щоб зберігати в пам'яті мільйони записів і сортувати їх наприкінці, застосовується алгоритм обмеженої черги. Система утримує в оперативній пам'яті лише сотню найбільших файлів; при знаходженні нового об'єкта, розмір якого перевищує найменший елемент у цій черзі, відбувається миттєва заміна. Такий метод дозволяє оптимізувати використання ресурсів процесора та пам'яті, підготовлюючи отримані статистичні метрики для збереження в локальній базі даних.

1.4. Методи пошуку дублікатів файлів

Проблема надлишковості збережених даних тісно пов'язана з наявністю дублікатів – файлів, які мають абсолютно ідентичний вміст, але можуть знаходитися в різних директоріях. Для їх виявлення існують різні алгоритмічні підходи, які суттєво відрізняються за швидкістю роботи та точністю результатів.

Найпростішим методом є порівняння файлів за їхніми іменами. Цей підхід не потребує читання фізичного вмісту і спирається виключно на системні метадані файлової системи. Однак на практиці він є вкрай ненадійним. З одного боку, файли з однаковими назвами (наприклад, текстові

документи з назвою «звіт») у різних робочих папках часто містять зовсім іншу інформацію. З іншого боку, справжні копії одного й того ж документа можуть мати змінені імена через автоматичне додавання суфіксів операційною системою під час копіювання [12]. Тому цей метод не може використовуватися як надійний критерій для пошуку.

Порівняння за розміром є значно ефективнішим інструментом первинної фільтрації. Якщо два об'єкти мають різний розмір з точністю до байта, вони гарантовано не є копіями [13]. Зчитування розміру з метаданих виконується миттєво, що дозволяє швидко відсіяти більшість унікальних даних. Проте збіг розмірів лише вказує на ймовірність дублювання, адже два різні медіафайли або архіви можуть випадково мати однаковий обсяг.

Для стовідсоткової впевненості в ідентичності необхідно аналізувати вміст файлів безпосередньо. Оскільки побайтове порівняння масивів даних великого обсягу є надто повільним і потребує значних витрат оперативної пам'яті, у сучасній практиці застосовують криптографічні хеш-функції. Хешування перетворює вхідний масив байтів довільної довжини на унікальний рядок фіксованого розміру (хеш-суму). Якщо вміст двох файлів повністю збігається, їхні хеш-суми будуть однакові.

Серед алгоритмів хешування найвідомішими є MD5 та SHA-256. Алгоритм MD5 тривалий час був стандартом завдяки високій швидкості обчислень, проте на сьогодні він вважається застарілим через виявлені вразливості до колізій – ситуацій, коли математично різні вхідні дані генерують однакову контрольну суму. Натомість стандарт SHA-256 формує значно довший підпис обсягом 256 біт, що зводить ймовірність випадкової колізії до нескінченно малої величини, гарантуючи унікальність результату для кожного унікального набору даних [12].

Враховуючи ці особливості, у розроблюваному додатку реалізовано комбінований архітектурний підхід. На першому етапі система групує всі знайдені файли за їхнім точним обсягом. Для тих груп, де виявлено більше одного об'єкта, запускається процес асинхронного читання вмісту з

подальшим обчисленням хеш-сум за алгоритмом SHA-256. Це дозволяє досягти абсолютної точності виявлення істинних дублікатів, паралельно мінімізуючи навантаження на систему, оскільки ресурсомісткому процесу хешування підлягають лише потенційні дублікати, а не весь масив просканиваних даних.

1.5. Методи візуалізації використання дискового простору

Аналіз великих масивів даних втрачає свою практичну цінність, якщо результати неможливо зручно інтерпретувати [14]. Сприйняття "сирих" числових показників розмірів тисяч файлів є надзвичайно складним для людини, тому графічний інтерфейс системи аналізу має спиратися на перевірені методи візуалізації просторової інформації [15].

Найбільш структурованим підходом до відображення точних характеристик є табличне представлення. Цей метод ідеально підходить для формування плоских списків, де кожен елемент має набір чітких атрибутів: шлях, обсяг, дату створення або хеш-суму. У розроблюваному програмному рішенні табличні компоненти застосовуються для виведення сотні найбільших об'єктів і згрупованих дублікатів [16]. Завдяки можливості швидкого сортування стовпців користувач може миттєво знаходити найважливі елементи без необхідності переглядати всю ієрархію, при цьому ширина колонок автоматично підлаштовується під розмір вмісту для максимальної читабельності.

Для відображення самої ієрархії традиційно використовують дерево каталогів. Цей метод візуалізації максимально наближений до стандартного досвіду взаємодії з файловим менеджером операційної системи. Вузли дерева відображають вкладеність папок, а поруч із їхніми назвами виводяться обчислені розміри. Як зазначалося вище, для забезпечення плавності інтерфейсу відображення такого дерева базується на концепції відкладеного завантаження графічних елементів. Дані про підкаталоги генеруються в дереві виключно в момент розгортання конкретного вузла користувачем.

Коли виникає потреба оцінити частку певних категорій даних у загальному обсязі (наприклад, розподіл за типами файлів), найбільш наочним інструментом є кругові діаграми. Вони дозволяють швидко сприйняти пропорції всього сканованого диска. Програмна реалізація цього методу в додатку виконана за допомогою базового графічного інтерфейсу (GDI+). Замість використання важких сторонніх бібліотек, система самостійно розраховує координати та промальовує сегменти діаграми, паралельно генеруючи адаптивну легенду, яка автоматично прокручується при великій кількості знайдених розширень файлів.

Альтернативним і досить популярним у спеціалізованому програмному забезпеченні методом є візуалізація treemap (деревоподібна карта). Вона перетворює ієрархічну структуру каталогів на набір вкладених прямокутників, площа яких строго пропорційна розміру файлів [17]. Незважаючи на високу інформативність цього методу під час пошуку великих об'єктів у глибоких директоріях, він має суттєвий недолік – складність сприйняття для непідготовленого користувача за великої кількості дрібних файлів, що візуально перетворює карту на нерозбірливу мозаїку. З огляду на це, архітектура розроблюваного додатку свідомо уникає treemap-візуалізації, віддаючи перевагу більш звичній комбінації інтерактивного дерева та кругових діаграм, що забезпечує баланс між інформативністю та простотою графічного інтерфейсу.

1.6. Аналіз існуючих програмних засобів та короткі висновки

Проектування конкурентоспроможного програмного забезпечення вимагає попереднього дослідження існуючих ринкових рішень для виявлення їхніх архітектурних недоліків та вдалих інтерфейсних знахідок. У сегменті утиліт для аналізу дискового простору для операційних систем сімейства Windows представлено значну кількість інструментів, серед яких найбільшої популярності набули WinDirStat, SpaceSniffer та TreeSize Free.

Додаток WinDirStat [18] є одним із найстаріших інструментів у цій категорії. Його головною особливістю є використання методу деревоподібних карт для візуалізації кожного знайденого файлу у вигляді кольорового прямокутника, розмір якого пропорційний реальному обсягу даних на диску. Незважаючи на високу деталізацію, такий підхід створює серйозне візуальне перевантаження. Крім того, архітектура програми є застарілою: процес сканування виконується в одному синхронному потоці, що призводить до тимчасового блокування графічного інтерфейсу на сучасних багатоядерних системах. Також ця утиліта не має вбудованих механізмів криптографічного порівняння вмісту, тому автоматичний пошук істинних дублікатів у ній відсутній.

Утиліта SpaceSniffer [19] також використовує блокову візуалізацію, проте пропонує більш динамічний підхід з анімацією процесу сканування в реальному часі. Програма є портативною і не потребує глибокої інтеграції в систему. Однак її інтерфейс повністю позбавлений традиційного табличного або ієрархічного представлення даних. Відсутність чітких списків найбільших файлів чи згрупованої статистики за типами даних суттєво ускладнює проведення швидкого аудиту. Додаток генерує виключно графічну карту, покладаючи завдання з виявлення надлишкових масивів інформації виключно на гостроту зору користувача.

Більш сучасним та наближеним до класичних файлових менеджерів є рішення TreeSize Free [20]. Воно використовує звичне дерево каталогів, що забезпечує інтуїтивно зрозумілу навігацію, а також демонструє високу швидкість обходу директорій. Головним недоліком цього додатка є агресивне штучне обмеження функціоналу в базовій безкоштовній версії. Зокрема, користувач не має можливості шукати точні копії файлів, будувати повноцінні графіки розподілу форматів та зберігати історію сканувань для ретроспективного аналізу використання пам'яті.

Аналіз розглянутих аналогів підтверджує практичну доцільність розробки власного програмного рішення. Існуючі утиліти або мають

перевантажений і технічно застарілий інтерфейс, або приховують критично важливі функції за платними ліцензіями. Проектована система аналізу безпосередньо вирішує ці проблеми: вона відмовляється від хаотичних деревоподібних карт на користь чистого ієрархічного дерева та зрозумілих кругових діаграм, використовує сучасні патерни асинхронного програмування для збереження плавності роботи під час глибокого сканування, а також об'єднує в єдиному просторі точний пошук дублікатів за хешем та автоматичне збереження статистики у локальній реляційній базі даних.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ АНАЛІЗУ ФАЙЛОВОЇ СИСТЕМИ

2.1 Постановка задачі дослідження та формування вимог

Аналіз існуючих аналогів продемонстрував, що розробка ефективного програмного забезпечення для аудиту накопичувачів вимагає визначення меж проєкту та базових сценаріїв використання. Головною задачею практичної частини є проєктування десктопного додатка для платформи Windows, який автоматизує процес пошуку надлишкових даних. Для успішного вирішення цієї проблеми програмне рішення має охоплювати чотири напрямки: аналіз структури диска, пошук великих файлів, пошук дублікатів і візуалізацію результатів. Відповідно до цих напрямів формується основний пул функціональних вимог.

Першочерговим завданням є аналіз структури диска. Додаток повинен рекурсивно обходити вкладені каталоги обраного накопичувача, підраховувати розмір кожної гілки та збирати статистику про типи знайдених об'єктів. Оскільки в операційній системі існують захищені директорії з обмеженими правами доступу, критичною вимогою є забезпечення безперервності сканування. Програма має програмно перехоплювати помилки читання (зокрема винятки доступу) та продовжувати роботу з доступними каталогами без аварійного завершення [21].

Другий аспект — пошук великих файлів. Замість того, щоб навантажувати оперативну пам'ять збереженням інформації про всі документи для подальшого сортування наприкінці сканування, система повинна діяти проактивно. Потрібно реалізувати алгоритм, який "на льоту" формуватиме та постійно оновлюватиме список зі ста найбільших об'єктів. Це дасть змогу користувачеві одразу після завершення обходу побачити файли, які займають найбільше місця.

Третьою важливою вимогою є пошук дублікатів. Просте порівняння за іменами чи збігом метаданих є недостатнім. Система має реалізувати алгоритм групування об'єктів з однаковим розміром та подальшого побайтового порівняння їхнього фізичного вмісту за допомогою надійного криптографічного алгоритму SHA-256. Це гарантує виявлення істинних копій.

Завершальним етапом виступає візуалізація результатів. Усі зібрані метрики потрібно подати у зручному графічному вигляді. Вимоги до інтерфейсу передбачають створення ієрархічного дерева каталогів із підтримкою ледачого завантаження для економії ресурсів, таблиць із можливістю швидкого сортування, а також кругових діаграм, які наочно демонструватимуть розподіл обсягу пам'яті між різними форматами даних.

Окрім функціональної частини, необхідно дотримуватися архітектурних та нефункціональних вимог. Базовою платформою є середовище .NET 8.0, а користувацький інтерфейс будується на основі Windows Forms. Для того, щоб програма швидко реагувала на дії користувача під час тривалих операцій з диском, вся логіка має виконуватися у фонових потоках з використанням асинхронних патернів. Збереження історії сканувань і статистичних показників забезпечуватиметься шляхом інтеграції з локальною базою даних SQLite.

2.2 Проєктування архітектури програмного засобу

Для забезпечення надійності та простоти подальшого супроводу розроблюваного програмного рішення було обрано багат шарову архітектуру з елементами патерна MVC [22]. Такий підхід дозволяє чітко розмежувати відповідальність між відображенням даних, бізнес-логікою та взаємодією з фізичною пам'яттю. Загальна структура системи поділяється на чотири ключові компоненти: модулі сканування, аналізу, візуалізації та роботи з базою даних.

На рисунку 2.1 наведено функціональну схему додатка для аналізу структури файлової системи та візуалізації використання дискового простору.

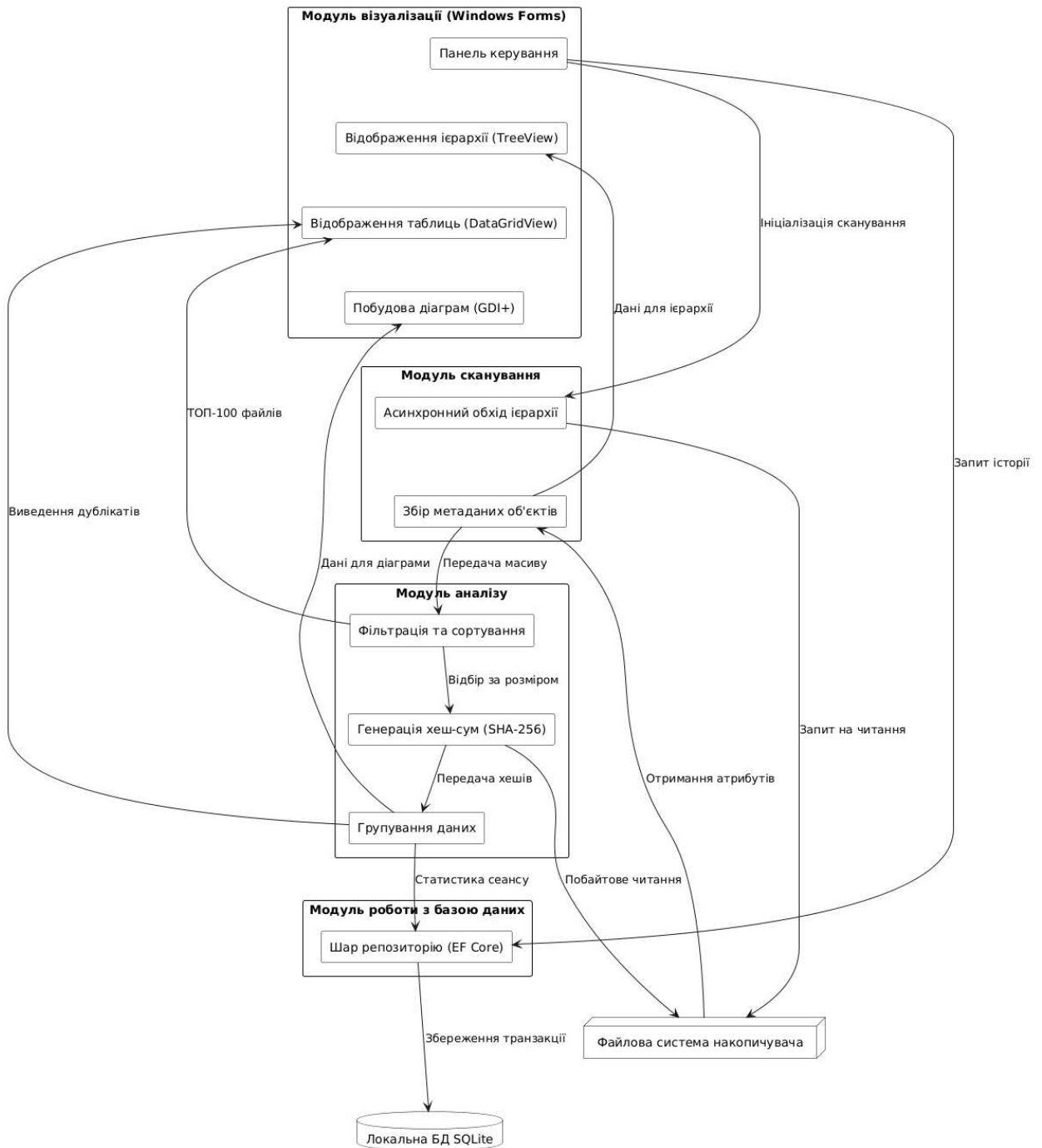


Рисунок 2.1 - Функціональна схема додатка для аналізу структури файлової системи та візуалізації використання дискового простору

Модуль сканування реалізовано на рівні сервісів бізнес-логіки. Його головна задача – рекурсивний обхід цільового накопичувача та збір інформації про кожен знайдений об'єкт. Для забезпечення високої швидкодії замість стандартного завантаження повних масивів файлів у пам'ять використовується

оптимізоване потокове перелічування через базові класи введення-виведення платформи .NET. Процес виконується асинхронно у фоновому потоці, паралельно генеруючи структури даних (моделі файлів і каталогів), які далі передаються по конвеєру. Цей модуль також відповідає за обробку помилок доступу, гарантуючи, що додаток не зупинить перевірку всього диска через одну ізольовану системну директорію.

Модуль аналізу відповідає за обробку зібраного набору даних. Він отримує готові моделі від сервісу сканування та застосовує інтегровані мовні запити для фільтрації, сортування та групування об'єктів. Саме тут функціонує алгоритм підтримки фіксованого списку найбільших файлів та логіка виявлення дублікатів. Сервіс аналізу спочатку відбирає групи об'єктів з однаковим розміром, після чого ініціює для них процедуру побайтового порівняння шляхом криптографічного хешування за алгоритмом SHA-256. Крім того, модуль агрегує статистику за розширеннями, розраховуючи кількісні та об'ємні показники для подальшого графічного представлення.

Модуль візуалізації є частиною користувацького інтерфейсу і відповідає за відображення обробленої інформації. Він побудований на класичних віконних компонентах, які просторово організовані за допомогою системи вкладок і розділених контейнерів. Ієрархія накопичувача виводиться у вигляді дерева з підтримкою відкладеного завантаження підкаталогів, а списки великих об'єктів і знайдених копій формуються в таблицях з автоматичним масштабуванням ширини колонок. Окремим підмодулем є сервіс побудови графіків, який використовує базовий графічний інтерфейс операційної системи для кастомного промальовування кругових діаграм. Він програмно обчислює кути секторів на основі відсоткового співвідношення типів файлів та генерує адаптивну легенду без використання важких сторонніх бібліотек.

Модуль роботи з базою даних ізолює всі операції запису та читання, реалізуючи патерн репозиторію. Для зберігання історії сканувань використовується локальна реляційна база даних SQLite, з якою взаємодіють за допомогою інструментів об'єктно-реляційного відображення (Entity

Framework Core). Модуль автоматично створює файл бази даних у локальній директорії під час першого запуску, керує схемою таблиць та забезпечує цілісність збереження масивів статистики шляхом виконання операцій у межах захищених транзакцій.

2.3 Проєктування структури даних

Запроєктована багатошарова архітектура вимагає створення фундаменту у вигляді об'єктних моделей, які циркулюватимуть між сервісами бізнес-логіки, графічним інтерфейсом та рівнем доступу до даних. У розроблюваному програмному рішенні абстракція фізичної файлової системи реалізується через набір взаємопов'язаних класів, кожен із яких відповідає за збереження специфічного набору інформації.

Основною інформаційною одиницею є модель `FileItem`. Ця структура інкапсулює ключові характеристики окремого документа: повний шлях у системі, назву, обсяг у байтах, розширення та дати створення й останньої модифікації. Для забезпечення роботи алгоритмів пошуку дублікатів у моделі передбачено спеціальне поле для зберігання криптографічної хеш-суми, яке заповнюється асинхронно лише для тих об'єктів, що підозрюються на ідентичність. Оскільки ці об'єкти безпосередньо прив'язуються до табличних компонентів інтерфейсу, модель також містить допоміжні властивості для форматування точних розмірів у зручний для сприйняття вигляд (кілобайти, мегабайти тощо).

Для представлення ієрархії каталогів розроблено модель `DirectoryItem`. Її спроектовано з використанням класичного об'єктно-орієнтованого патерну «Компонувальник», що дозволяє алгоритмам однаково працювати як з поодинокими файлами, так і з їхніми розгалуженими сукупностями. Кожен екземпляр цієї моделі містить дві внутрішні колекції: список безпосередньо вкладених об'єктів `FileItem` та список дочірніх підкаталогів. Головною архітектурною особливістю цієї моделі є динамічний розрахунок розміру: загальний обсяг директорії не зчитується безпосередньо з метаданих

операційної системи, а рекурсивно підсумовується на основі розмірів усіх її внутрішніх елементів.

Під час роботи модуля аналізу виникає потреба в агрегуванні знайдених копій. Для цього створено структуру `DuplicateGroup`. На відміну від попередніх сутностей, вона не відображає реальний фізичний об'єкт на диску, а виступає логічним контейнером для групування файлів з підтвердженим однаковим вмістом. Ця модель утримує загальний розмір одного екземпляра, лічильник знайдених копій та масив посилань на конкретні об'єкти `FileItem`. На основі цих даних структура має здатність автоматично розраховувати метрику надлишкового простору – обсяг пам'яті, який гарантовано звільниться у разі видалення всіх копій, окрім одного оригіналу.

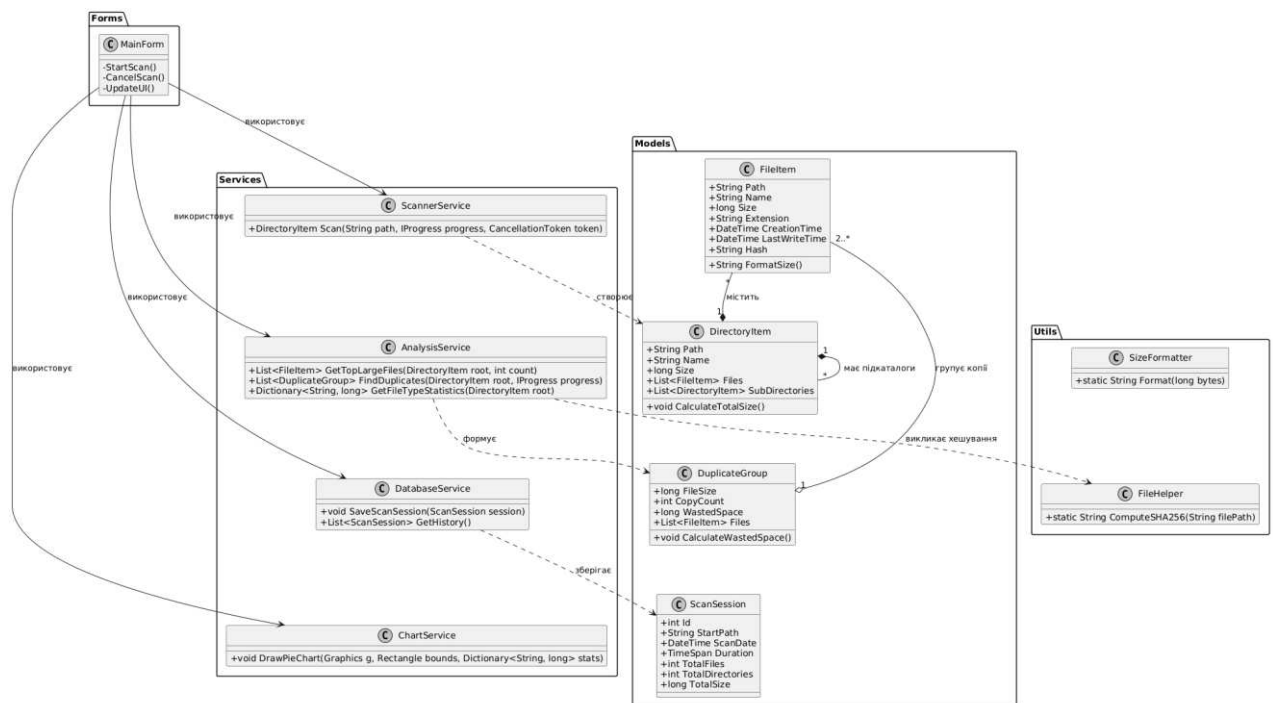


Рисунок 2.2 – Діаграма класів додатку для аналізу структури файлової системи та візуалізації використання дискового простору

Для збереження результатів сканування в локальній базі даних використовується коренева сутність `ScanSession`. Вона виконує роль основної моделі для системи об'єктно-реляційного відображення і фіксує історичний

зріз стану накопичувача на момент завершення перевірки. Модель агрегує в собі основні показники: стартовий шлях перевірки, мітку дати та часу, витрачений час на сканування, а також загальну кількість знайдених файлів і каталогів. Ізоляція цієї інформації в окрему сутність дозволяє швидко завантажувати історію попередніх сеансів, надаючи користувачеві можливість відстежувати динаміку зміни вільного простору без запуску ресурсомісткого процесу повторного сканування.

2.4 Проектування бази даних

Для збереження історії перевірок та зібраної статистики розроблена система потребує надійного механізму тривалого зберігання інформації. Використання повноцінних клієнт-серверних систем керування базами даних є недоцільним для локального настільного додатку, оскільки це вимагає від користувача додаткових зусиль зі встановлення та налаштування стороннього програмного забезпечення. Тому було прийнято рішення інтегрувати компактну реляційну базу даних SQLite. Вона функціонує як звичайний локальний файл і не потребує запуску окремого фонових процесу чи служби. База даних автоматично генерується під час першого запуску програми та розміщується у стандартній прихованій директорії профілю операційної системи, призначеній для локальних даних додатків користувача.

Взаємодія з базою даних на рівні програмного коду реалізована за допомогою сучасних інструментів об'єктно-реляційного відображення. Це дозволяє працювати з таблицями так само, як зі звичайними об'єктними моделями, повністю абстрагуючись від ручної генерації текстових SQL-запитів. Оскільки збереження результатів глибокого сканування передбачає одночасний запис десятків, а іноді й сотень тисяч рядків, усі операції виконуються в межах єдиної транзакції. Такий підхід гарантує цілісність інформації: у разі раптового збою, вимкнення живлення або примусового закриття програми неповні дані просто скасовуються, залишаючи файл бази даних у стабільному стані.

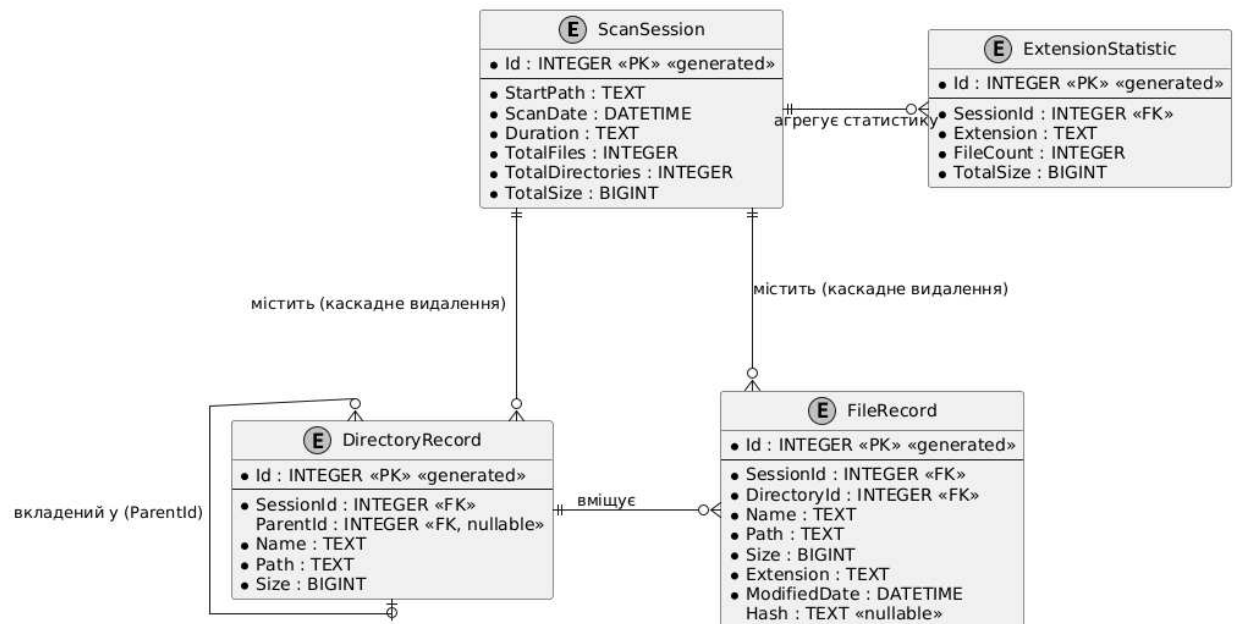


Рисунок 2.3 – Схема бази даних додатку для аналізу структури файлової системи та візуалізації використання дискового простору

Структура збереження даних базується на кількох взаємопов'язаних таблицях, які точно відображають ключові моделі предметної області. Головною таблицею є реєстр сеансів сканування. Кожен запис у ній фіксує унікальний первинний ключ перевірки, початковий шлях на диску, точну дату та час початку, витрачений час, а також загальні підсумкові обсяги. Ця таблиця слугує основною точкою входу для відображення історичних даних у графічному інтерфейсі.

Навколо реєстру сеансів будуються підпорядковані таблиці, які зберігають деталізовану інформацію про кожну перевірку. Таблиця файлів містить повні шляхи, назви, розміри в байтах, розширення та дати модифікації всіх знайдених документів. Кожен такий запис за допомогою зовнішнього ключа прив'язаний до конкретного ідентифікатора сеансу сканування.

Аналогічно функціонує таблиця каталогів, проте вона має специфічну внутрішню структуру. Для відтворення багаторівневої ієрархії файлової системи безпосередньо в базі даних реалізовано рекурсивний зв'язок: кожен запис містить додатковий ідентифікатор, який вказує на батьківський каталог

у цій самій таблиці. Окремо виділено таблицю для зберігання згрупованої статистики. Вона агрегує дані про кожне виявлене розширення файлу, зберігаючи кількість таких документів та їхній сумарний обсяг у межах конкретної перевірки.

Між усіма сутностями встановлено реляційні зв'язки. Таблиця сеансів має відношення «один до багатьох» з таблицями файлів, каталогів та статистичних зведень. Це архітектурне рішення забезпечує автоматичне каскадне видалення інформації. Якщо користувач вирішує очистити історію певної перевірки диска для звільнення пам'яті, система гарантовано видаляє всі пов'язані файли, папки та метрики, запобігаючи накопиченню «осиротілих» та незв'язаних записів.

2.5 Алгоритм роботи програми

Під час створення логічного ядра програмного рішення основна увага приділялася оптимізації процесу обробки великих масивів інформації. Оскільки сучасні накопичувачі можуть містити мільйони різноманітних файлів, використання класичного підходу зі збереженням усіх даних у пам'ять для подальшого аналізу є неефективним. Робота розробленого алгоритму розпочинається з етапу ініціалізації, під час якого програма готує необхідні внутрішні масиви для збереження статистики та переносить усе обчислювальне навантаження у фоновий режим. Це дозволяє уникнути зависання графічного вікна під час тривалих операцій читання жорсткого диска.

Основний цикл алгоритму побудований на послідовному рекурсивному обході структури папок. Заходячи в кожен новий каталог, програма спершу перевіряє наявність прав доступу. У разі спроби сканування захищених операційною системою директорій неминуче виникає помилка, проте завдяки застосуванню принципу м'якої деградації алгоритм просто фіксує факт обмеження доступу та спокійно переходить до наступної відкритої гілки, продовжуючи загальний процес сканування без переривань.

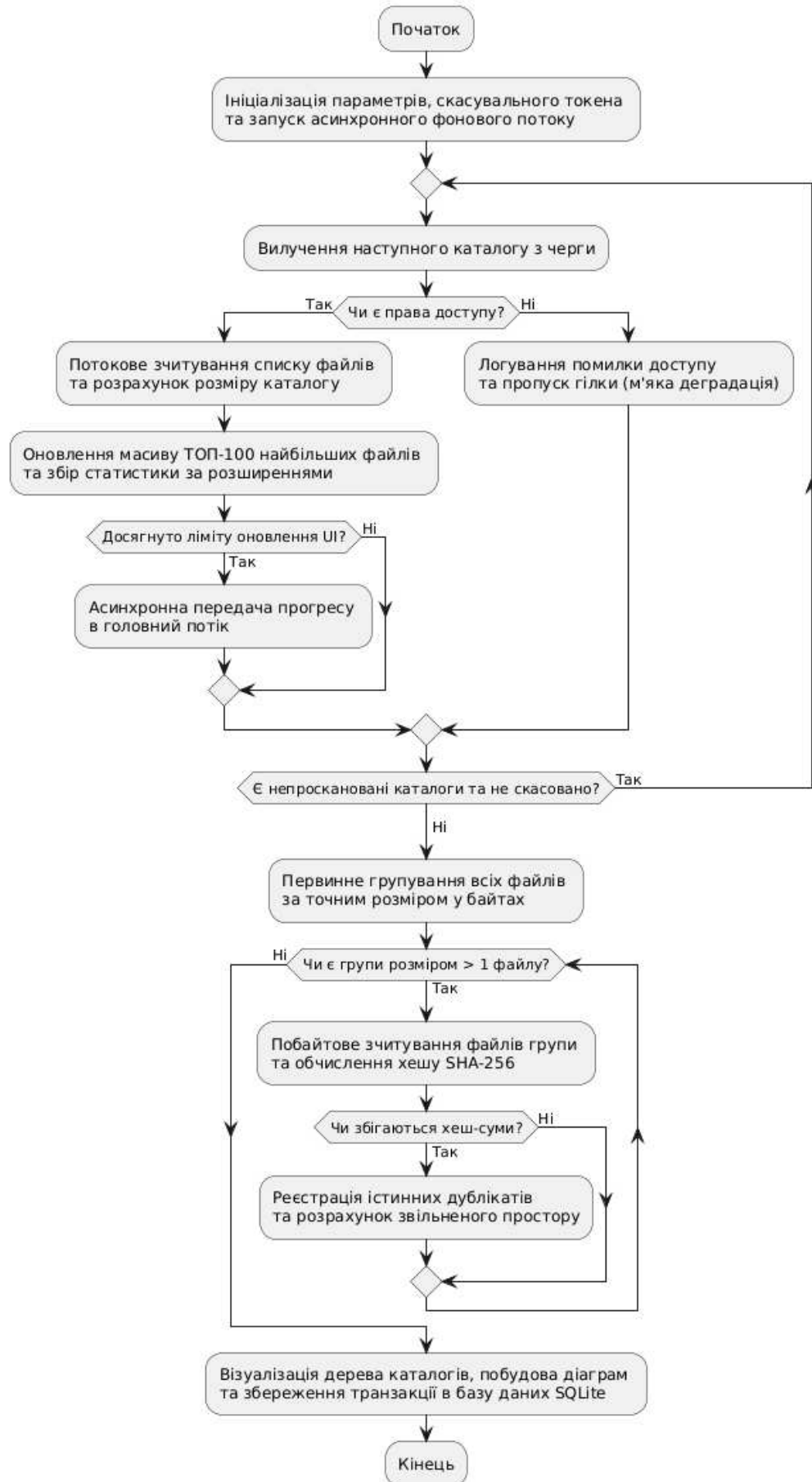


Рисунок 2.4 – Алгоритм роботи програми

Успішно відкривши каталог, система зчитує перелік розташованих у ньому документів і відразу додає їхні обсяги до загального розміру поточної папки. Одночасно з цим на льоту оновлюється список зі ста найбільших знайдених файлів. Коли на диску трапляється новий об'ємний документ, алгоритм порівнює його розмір із найменшим елементом у списку лідерів, виконуючи миттєву заміну за потреби. Також програма безперервно збирає статистику розширень, підраховуючи кількість і розмір об'єктів кожного формату. Щоб користувач міг орієнтуватися в часі перевірки, інформація про хід аналізу періодично транслюється на головний екран дозованими порціями, що оберігає інтерфейс від перенавантаження.

Після завершення обходу ієрархії починається другий ключовий етап – виявлення надлишкових даних. Для заощадження процесорного часу алгоритм спочатку групує абсолютно всі зібрані файли за їхнім точним розміром у байтах. Унікальні за обсягом документи миттєво відсіюються, оскільки вони не можуть бути копіями за законами фізики носіїв інформації. Для тих груп, де знайдено два або більше елементів однакового розміру, запускається процес поглибленого зчитування вмісту з обчисленням криптографічної хеш-суми. Об'єкти з повністю ідентичними підписами маркуються як справжні дублікати, після чого розраховується обсяг дискового простору, який займають ці зайві копії.

На завершальній стадії алгоритм агрегує всі обчислені показники. Згенеровані масиви інформації передаються до графічного модуля, який розгортає візуальне дерево каталогів та будує кругові діаграми розподілу форматів. Паралельно з відмальовуванням інтерфейсу вся зібрана структура результатів зберігається до локальної бази даних у межах єдиної безпечної транзакції, надійно фіксуючи історію сеансу для її подальшого перегляду в майбутньому.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ АНАЛІЗУ ФАЙЛІВ

3.1 Обґрунтування вибору засобів розробки

Перехід від етапу теоретичного проектування до безпосереднього написання коду вимагає чіткого визначення технологічного стеку. Вибір інструментарію має базуватися не лише на сучасних тенденціях, а й на специфіці самого програмного рішення – десктопної утиліти для інтенсивної роботи з локальною файловою системою.

Основною мовою програмування було обрано C#, яка функціонує в середовищі актуальної платформи .NET 8. Вибір C# зумовлений її суворою типізацією, об'єктно-орієнтованою природою та вбудованими механізмами для безпечної роботи з пам'яттю. Для розроблюваного додатка критично важливою є наявність нативних мовних конструкцій `async/await`. Вони дозволяють легко реалізувати багатопотокове сканування дисків, не блокуючи роботу інтерфейсу під час важких дискових операцій. Своєю чергою, використання рантайму .NET 8 забезпечує високу продуктивність базових класів вводу-виводу (зокрема `System.IO Enumeration`), що є принципово важливим під час обходу каталогів з мільйонами дрібних файлів.

Для побудови графічного інтерфейсу користувача було вирішено використати класичний фреймворк Windows Forms. Хоча сьогодні існують більш сучасні інструменти (наприклад, WPF або MAUI), Windows Forms залишається оптимальним вибором для створення невеликих системних утиліт під ОС Windows. Ця технологія надає прямий доступ до стандартних системних компонентів, таких як `TreeView` для відображення дерева каталогів та `DataGridView` для формування таблиць найбільших файлів і дублікатів. Такий підхід гарантує, що додаток виглядатиме органічно в середовищі операційної системи та споживатиме мінімальну кількість оперативної пам'яті для відмальовування інтерфейсу.

Збереження історії сеансів перевірок та зібраної статистики потребує використання бази даних. Розгортання повноцінних клієнт-серверних систем (на зразок SQL Server або MySQL) є абсолютно недоцільним для локальної десктопної утиліти, оскільки це невиправдано ускладнює процес встановлення програми для кінцевого користувача. Ідеальним рішенням стала інтеграція SQLite. Це безсерверна, легка реляційна база даних, яка зберігає всю інформацію в одному локальному файлі безпосередньо в папці профілю користувача. Вона не потребує жодних попередніх налаштувань і забезпечує надійне збереження десятків тисяч записів у межах захищених транзакцій.

Незважаючи на те, що базові засоби графічного інтерфейсу дозволяють самостійно малювати геометричні фігури, ручне програмування складних діаграм забирає багато часу і часто поступається якістю готовим рішенням. Тому для візуалізації статистики за типами файлів було інтегровано сторонню бібліотеку LiveCharts [22]. Вона дозволяє швидко генерувати інтерактивні кругові діаграми з автоматичним розрахунком пропорцій секторів. Завдяки LiveCharts користувач може наводити курсор на окремі сегменти графіка та отримувати миттєві підказки щодо точного розміру певного формату файлів, що значно підвищує загальну ергономіку розробленого програмного засобу.

3.2 Реалізація функціональних можливостей програмного засобу

Відповідно до спроектованої архітектури, процес написання коду було розділено на логічні блоки, кожен з яких відповідає за конкретний аспект обробки даних. Щоб уникнути перевантаження вихідного коду, реалізацію виконано з дотриманням принципів об'єктно-орієнтованого програмування, де кожен сервіс виконує чітко визначену задачу. Далі наведено короткий опис реалізацій.

3.2.1 Реалізація сканування та аналізу даних

Функціонал обходу каталогів побудовано на основі методів простору імен System.IO. Для економії оперативної пам'яті замість традиційного методу

GetFiles, який завантажує весь масив даних одночасно, використано EnumerateFiles. Цей метод працює як ітератор, повертаючи інформацію про файли по черзі. Оскільки під час сканування системного диска неминуче виникають конфлікти прав доступу, рекурсивний алгоритм обгорнуто у блок try-catch. Перехоплення винятку UnauthorizedAccessException дозволяє програмі просто ігнорувати закриті директорії та продовжувати роботу з доступними даними.

Аналіз розмірів папок виконується динамічно під час рекурсивного згортання: щойно алгоритм досягає найглибшого підкаталогу, він підсумовує обсяги всіх його файлів і передає це значення на рівень вище. Паралельно з цим розширення кожного знайденого документа фіксується в колекції типу Dictionary<string, long>, де ключем є формат файлу, а значенням – його сумарний розмір.

3.2.2 Програмна реалізація пошуку дублікатів

Для виявлення істинних копій реалізовано двоступеневий алгоритм з використанням мовних запитів LINQ. На першому етапі весь масив просканованих об'єктів групується за властивістю Length (точний розмір у байтах). Усі унікальні за розміром файли відкидаються. Для груп, що містять два і більше елементів, ініціюється другий етап – криптографічне порівняння. За допомогою класу SHA256 програма побайтово зчитує вміст файлів та генерує унікальні хеш-суми. Якщо хеші повністю збігаються, файли об'єднуються в об'єкт моделі DuplicateGroup, для якої автоматично розраховується потенційно звільнений простір (шляхом множення розміру файлу на кількість зайвих копій).

3.2.3 Взаємодія з базою даних SQLite

Збереження історії перевірок реалізовано за допомогою фреймворку Entity Framework Core. У коді створено клас контексту DbContext, який описує структуру бази даних. Під час першого запуску програми викликається метод

Database.EnsureCreated(), який автоматично генерує локальний файл БД DiskAnalyzer.db у директорії профілю користувача.

Після завершення сканування всі створені об'єктні моделі (сеанс, каталоги, файли та статистика) передаються в контекст. Виклик асинхронного методу SaveChangesAsync() автоматично транслює ці об'єкти в SQL-запити та виконує їх у межах єдиної транзакції. Завантаження минулих сеансів для перегляду реалізовано через прості LINQ-запити до відповідних таблиць БД, що гарантує швидке отримання історії без повторного сканування.

3.2.4 Побудова інтерфейсу та візуалізація

Користувацький інтерфейс скомпоновано за допомогою стандартних контейнерів Windows Forms. Відображення структури каталогів покладено на компонент TreeView. Щоб програма не зависала під час спроби одночасно відмалювати велику кількість вузлів, реалізовано патерн ледачого завантаження: інформація про підкаталоги додається до дерева візуально лише в момент спрацьовування події BeforeExpand (коли користувач натискає на іконку розгортання папки).

Табличні дані виводяться через компонент DataGridView з активованим режимом автоматичного підлаштування ширини колонок під контент. Візуалізація статистики за форматами файлів реалізована за допомогою сторонньої бібліотеки LiveCharts, яка генерує інтерактивні кругові діаграми з автоматичним розрахунком кутів секторів та виведенням відсоткового співвідношення.

3.3 Огляд роботи програми

Розроблений графічний інтерфейс користувача побудовано за принципом єдиного вікна з використанням вкладок, що забезпечує інтуїтивно зрозумілу взаємодію з програмою та швидкий доступ до всіх аналітичних даних. Робота з додатком передбачає лінійну послідовність дій, супроводжувану візуальним зворотним зв'язком на кожному етапі.

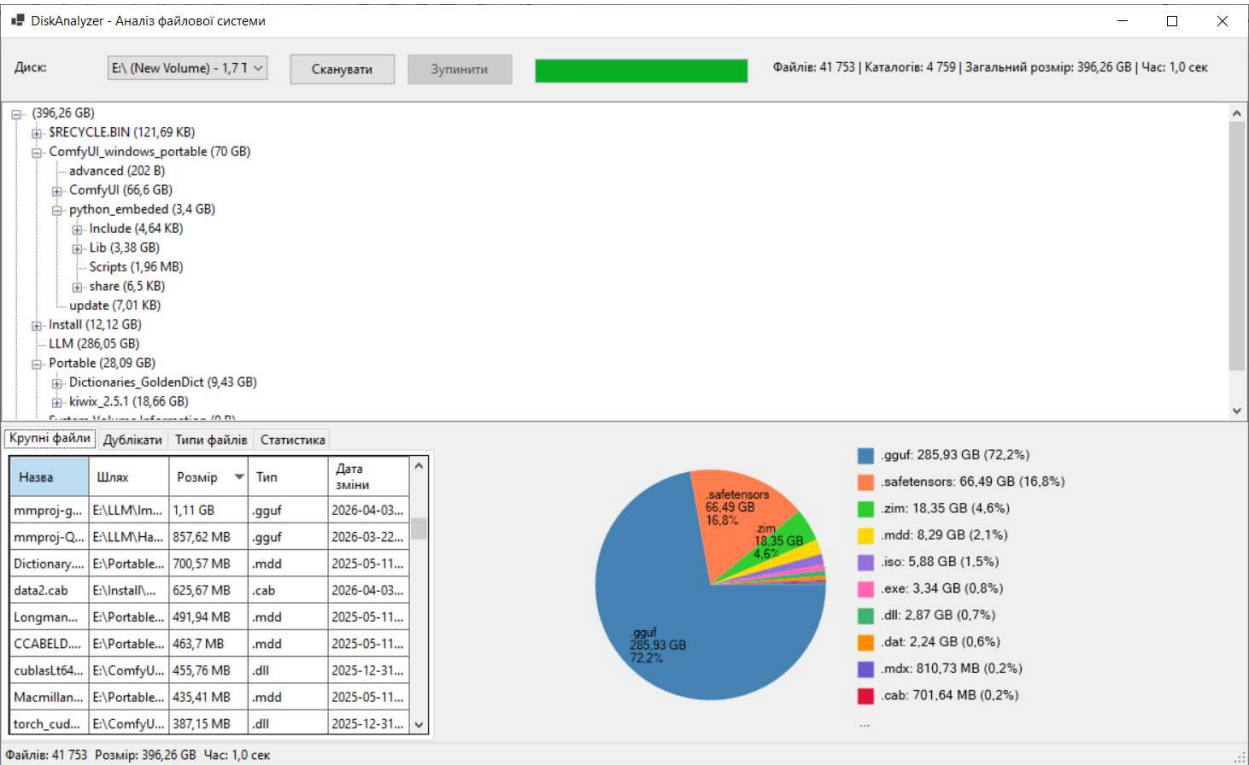


Рисунок 3.1 – Головне вікно програми для аналізу структури файлової системи та візуалізації використання дискового простору

Сценарій використання програмного засобу розпочинається з вибору об'єкта для аналізу. У верхній частині головного вікна розташована панель керування, яка містить поле для вибору поточного диску та кнопку «Сканувати».

Після вибору необхідного логічного диску, користувач натискає кнопку «Сканувати». Програма блокує елемент вибору шляху, і розпочинає фоновий процес обходу файлової системи. Оскільки процес може займати тривалий час, на панелі керування активується кнопка «Скасувати». Її натискання надсилає сигнал, що дозволяє безпечно перервати операцію без зависання програми чи пошкодження бази даних.

Під час роботи алгоритму користувач спостерігає за елементами індикації стану у вигляді текстового індикатора та смуги прогресу. Текстовий індикатор динамічно відображає шлях до каталогу, який сканується в даний час, а також загальну кількість знайдених на цей момент файлів. Водночас

смуга прогресу переводиться у режим пульсації під час первинного обходу, оскільки точна кількість файлів на диску заздалегідь невідома, або ж заповнюється лінійно під час етапу побайтового хешування дублікатів.

Після успішного завершення сканування або його примусового скасування зібрана інформація автоматично розподіляється між тематичними вкладками компонента, який займає основну робочу область вікна. На вкладці «Структура диска» головним елементом є ієрархічне дерево, що візуально повторює структуру провідника Windows, проте поруч із назвою кожної папки у дужках виводиться її розмір. Завдяки механізму ледачого завантаження користувач може миттєво розгортати найглибші гілки каталогів без затримок інтерфейсу.

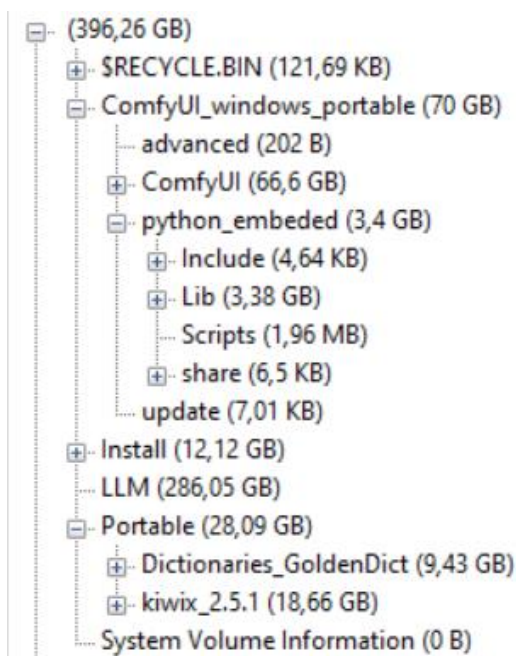


Рисунок 3.2 – Дерево структури диску

Вкладка «Крупні файли» представляє дані у вигляді інтерактивної таблиці з колонками для назви, шляху, розміру та дати зміни об'єктів. Користувач може натискати на заголовки колонок для прямого або зворотного сортування записів, що дозволяє швидко виявити найбільш об'ємні документи на диску.

Крупні файли					Дублікати	Типи файлів	Статистика
Назва	Шлях	Розмір	Тип	Дата зміни			
mmpoj-gemma-4...	E:\LLM\lmstudio-с...	1,11 GB	.gguf	2026-04-03 11:54:22			
mmpoj-Qwen3.5-...	E:\LLM\HauhauCS...	857,62 MB	.gguf	2026-03-22 14:27:59			
Dictionary.com 20...	E:\Portable\Diction...	700,57 MB	.mdd	2025-05-11 22:22:25			
data2.cab	E:\Install\OriginPro...	625,67 MB	.cab	2026-04-03 01:42:12			
LongmanDictionar...	E:\Portable\Diction...	491,94 MB	.mdd	2025-05-11 22:22:30			
CCABELD.mdd	E:\Portable\Diction...	463,7 MB	.mdd	2025-05-11 22:22:23			
cublasLt64_13.dll	E:\ComfyUI_windo...	455,76 MB	.dll	2025-12-31 05:49:32			
MacmillanEnEn.m...	E:\Portable\Diction...	435,41 MB	.mdd	2025-05-11 22:22:22			
torch_cuda.dll	E:\ComfyUI_windo...	387,15 MB	.dll	2025-12-31 05:49:42			
blaz3-unasfstage	E:\ComfyUI_windo...	330,64 MB	.mdd	2025-01-01 22:15:47			

Рисунок 3.3 – Вкладка «Крупні файли»

Наступна вкладка «Дублікати» також використовує табличне представлення, але зі специфічним групуванням, де в окремій текстовій панелі над таблицею виводиться сумарний обсяг надлишкового простору, який можна звільнити у разі видалення цих копій.

Крупні файли				Дублікати	Типи файлів	Статистика
Назва файлу	Кількість	Розмір файлу	Зайвий простір			
Qwen3.5-35B-A3B-Unce...	2	64,61 GB	64,61 GB			
_VF.pyi	2	1,17 MB	1,17 MB			
pip.exe	3	105,83 KB	211,66 KB			
_spdx.py	3	47,26 KB	94,53 KB			
LICENSE.APACHE	5	9,94 KB	39,74 KB			
image_z_image_turbo_f...	2	35,36 KB	35,36 KB			
LICENSE	2	34,33 KB	34,33 KB			
metadata.py	2	33,92 KB	33,92 KB			
LICENSE	4	11,09 KB	33,28 KB			

Рисунок 3.4 – Вкладка «Дублікати»

Вкладка «Типи файлів» зосереджена на аналізі категорій документів. У цьому табличному представленні наведено перелік усіх розширень файлів, виявлених під час сканування, кількість об'єктів для кожної категорії, а також їхній загальний обсяг.

Крупні файли Дублікати Типи файлів Статистика			
Розширення	Кількість файлів	Загальний розмір	Відсоток
.gguf	9	285,93 GB	72,16%
.safetensors	11	66,49 GB	16,78%
.zim	6	18,35 GB	4,63%
.mdd	19	8,29 GB	2,09%
.iso	1	5,88 GB	1,48%
.exe	56	3,34 GB	0,84%
.dll	138	2,87 GB	0,72%
.dat	31	2,24 GB	0,57%
.mdx	19	810,73 MB	0,20%

Рисунок 3.5 – Вкладка «Типи файлів»

Вкладка «Статистика» виконує роль головної інформаційної панелі, яка надає користувачеві зведені кількісні та часові метрики за результатами останнього сеансу сканування. У верхній частині цієї вкладки відображається загальна інформація про цільовий каталог або накопичувач, зокрема сумарний проаналізований обсяг, а також співвідношення зайнятого та вільного простору, що для наочності може супроводжуватися лінійним індикатором заповненості.

Центральна область відведена під статистику самого процесу обходу файлової системи. Тут користувач може побачити точну кількість просканованих підкаталогів і файлів, середній розмір одного об'єкта, а також сумарний обсяг виявлених надлишкових даних.

Окремим блоком виводиться інформація про продуктивність алгоритму, яка включає час початку операції та загальну тривалість перевірки у хвилинах і секундах.

Таке комплексне подання метрик дозволяє користувачеві оцінити як стан дискового простору, так і швидкодію самої програми без необхідності переходу до інших розділів додатка.

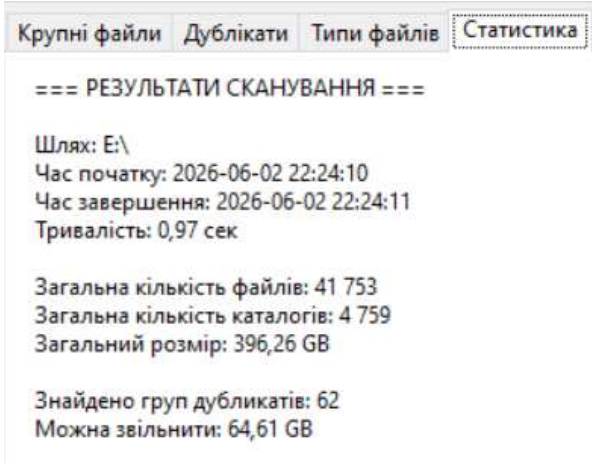


Рисунок 3.6 – Вкладка «Статистика»

Для більш наочної оцінки розподілу дискового простору між різними форматами табличні дані доповнюються інтерактивною круговою діаграмою, побудованою за допомогою графічної бібліотеки.

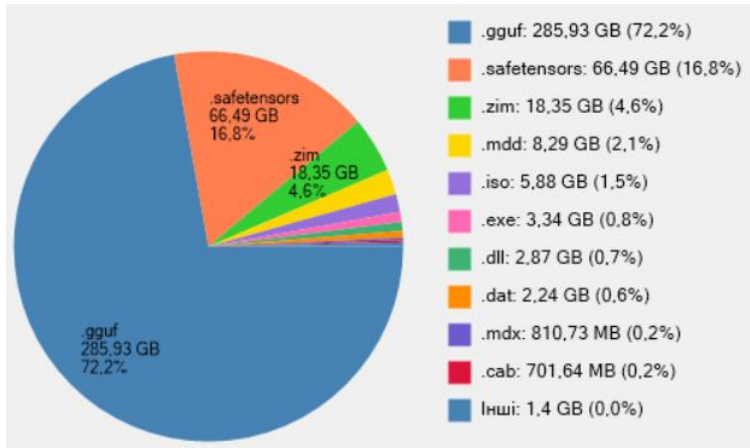


Рисунок 3.7 – Інтерактивна діаграма розподілу типів файлів на диску

Цей елемент дозволяє оперативно визначати, які саме типи даних найбільше перевантажують накопичувач. Розмір кожного сектора обчислюється автоматично і відповідає відсотковій частці певного формату в загальному масиві. Щоб уникнути візуального перевантаження інтерфейсу у випадках з великою різноманітністю розширень, алгоритм відбирає лише найбільш об'ємні категорії, тоді як решта дрібних форматів групується в єдиний зведений сектор.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розв'язано актуальне завдання з розробки спеціалізованого програмного забезпечення для глибокого аналізу файлової системи, виявлення надлишкових даних та візуалізації розподілу дискового простору. Під час проектування архітектури системи здійснено обґрунтований вибір сучасного технологічного стеку, основу якого склали мова програмування C# у середовищі .NET, класичний фреймворк Windows Forms та локальна безсерверна база даних SQLite. Такий підхід забезпечив оптимальний баланс між високою продуктивністю обчислень, стабільністю та зручністю розгортання десктопного додатка на кінцевих пристроях користувачів.

Важливим практичним результатом стала розробка оптимізованого алгоритму сканування, який реалізує механізм асинхронного рекурсивного обходу структури каталогів на базі потокового перелічування об'єктів. Впровадження принципу м'якої деградації для обробки системних помилок доступу дозволило уникнути аварійних зупинок алгоритму та суттєво знизити споживання оперативної пам'яті при аналізі накопичувачів великого обсягу. На основі цього алгоритму створено комплекс аналітичних модулів, зокрема підсистему динамічного формування списку найбільших файлів та ефективний двоступеневий механізм пошуку істинних дублікатів. Цей механізм базується на первинному відсіюванні документів за точним розміром у байтах із подальшим побайтовим порівнянням з використанням криптографічного алгоритму хешування SHA-256.

Отже, розроблений програмний засіб повністю відповідає поставленим завданням. Програма демонструє високу швидкодію, стійкість до помилок і може ефективно використовуватися для регулярного моніторингу файлової системи, виявлення критичних витрат пам'яті та безпечного звільнення дискового простору.

ПЕРЕЛІК ПОСИЛАНЬ

1. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. 10th ed. Hoboken : Wiley, 2018. 1216 p.
2. Meyer D. T., Bolosky W. J. A study of practical deduplication. ACM Transactions on Storage. 2011. Vol. 7, No. 4. P. 1–20.
3. Шеховцов В. А. Операційні системи : підручник. Київ, 2005. 576 с.
4. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Upper Saddle River : Pearson, 2014. 1136 p.
5. Arpaci-Dusseau R. H., Arpaci-Dusseau A. C. Operating Systems: Three Easy Pieces. Madison : Arpaci-Dusseau Books, 2018. 714 p.8.
6. Carrier B. File System Forensic Analysis. Boston : Addison-Wesley Professional, 2005. 600 p.
7. Задерейко О. В. Операційні системи : навчальний посібник [Електронне видання] / О. В. Задерейко, С. Л. Зіноватна, А. А. Толокнов. – Одеса : Фенікс, 2022. – 140 с
8. Overview of FAT, HPFS, and NTFS File Systems. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/troubleshoot/windows-client/backup-and-storage/fat-hpfs-and-ntfs-file-systems>
9. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 3rd ed. Cambridge : MIT Press, 2009. 1292 p.
10. Коротєєва Т. О. Алгоритми та структури даних : навчальний посібник. Львів : Видавництво Львівської політехніки, 2014. 280 с.
11. Richter J. CLR via C#. 4th ed. Redmond : Microsoft Press, 2012. 896 p.
12. Горбенко І. Д., Горбенко Ю. І. Прикладна криптологія. Теорія. Практика. Застосування : підручник. Харків : Форт, 2012. 879 с.
13. Ferguson N., Schneier B., Kohno T. Cryptography Engineering: Design Principles and Practical Applications. Indianapolis : Wiley, 2010. 384 p.
14. Few S. Show Me the Numbers: Designing Tables and Graphs to Enlighten. 2nd ed. Burlingame : Analytics Press, 2012. 371 p.

15. Shneiderman B., Plaisant C., Cohen M. Designing the User Interface: Strategies for Effective Human-Computer Interaction. 6th ed. Boston : Pearson, 2016. 624 p.
16. Грицюк Ю. І., Рак Т. Є. Програмування та розробка графічних інтерфейсів : підручник. Львів : Вид-во Львівської політехніки, 2019. 312 с.
17. Shneiderman B. Tree visualization with tree-maps: 2-d space-filling approach. ACM Transactions on Graphics. 1992. Vol. 11, No. 1. P. 92–99.
18. WinDirStat. WinDirStat : офіційний сайт. URL: <https://windirstat.net/>
19. SpaceSniffer. Uderzo Software : офіційний сайт. URL: http://www.uderzo.it/main_products/space_sniffer/
20. TreeSize Free. JAM Software : офіційний сайт. URL: https://www.jam-software.com/treesize_free/
21. Asynchronous programming in C#. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/>
22. LiveCharts2. Simple, flexible, interactive & powerful charts, maps and gauges. LiveCharts Official Documentation. URL: <https://livecharts.dev/>

Додаток А - Програмний код

ScannerService.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading;
using DiskAnalyzer.Models;
using DiskAnalyzer.Utills;
using DiskAnalyzer.Forms;

namespace DiskAnalyzer.Services
{
    /// <summary>
    /// Сервіс для сканування файлової системи
    /// Виконує рекурсивний обхід каталогів, збирає інформацію
    про файли та каталоги
    /// </summary>
    public class ScannerService
    {
        // Кількість файлів для відображення в TOP-списку
        private const int TOP_FILES_COUNT = 100;

        // Кількість каталогів для відображення в TOP-списку
        private const int TOP_DIRECTORIES_COUNT = 50;

        // Список для зберігання найбільших файлів
        private List<FileItem> _largestFiles;

        // Список для зберігання найбільших каталогів
        private List<DirectoryItem> _largestDirectories;

        // Словник для групування файлів за розширеннями
        private Dictionary<string, FileTypeStatistics>
        _fileTypeStatistics;

        // Загальна кількість файлів
        private long _totalFiles;

        // Загальна кількість каталогів
        private long _totalDirectories;

        // Загальний розмір
        private long _totalSize;

        // Лічильник каталогів з останнього оновлення прогресу
        private int _directoriesSinceLastProgress = 0;
```

```

// Кількість каталогів між оновленнями прогресу
private const int DIRECTORIES_PER_PROGRESS_UPDATE = 10;

/// <summary>
/// Конструктор сервісу сканування
/// </summary>
public ScannerService()
{
    _largestFiles = new List<FileItem>();
    _largestDirectories = new List<DirectoryItem>();
    _fileTypeStatistics = new Dictionary<string,
FileTypeStatistics>();
    _totalFiles = 0;
    _totalDirectories = 0;
    _totalSize = 0;
    _directoriesSinceLastProgress = 0;
}

/// <summary>
/// Сканує заданий каталог рекурсивно
/// </summary>
/// <param name="path">Шлях до каталогу для
сканування</param>
/// <param name="progress">Інтерфейс для звітування про
прогрес</param>
/// <param name="cancellationToken">Токен для скасування
сканування</param>
/// <returns>Результат сканування</returns>
public ScanResult ScanDirectory(
    string path,
    IProgress<ScanProgress>? progress = null,
    CancellationToken cancellationToken = default)
{
    try
    {
        // Очищення попередніх результатів
        ClearResults();

        Debug.WriteLine($"ScanDirectory: початок
сканування шляху: {path}");

        // Створення результату сканування
        ScanResult result = new ScanResult(path);

        // Перевірка існування шляху
        if (!Directory.Exists(path))
        {
            result.SetError($"Каталог не існує:
{path}");

            return result;
        }
    }
}

```



```

        // Отримання інформації про диск для оцінки
прогресу
        DriveInfo? driveInfo =
FileHelper.GetDriveInfo(path.Substring(0, 2));
        long totalDriveSize = driveInfo?.TotalSize ?? 1;

        // Рекурсивне сканування каталогу
DirectoryItem rootDirectory =
ScanDirectoryRecursive(
    path,
    null,
    progress,
    cancellationToken,
    totalDriveSize);

        // Оновлення результату
result.RootDirectory = rootDirectory;
result.TotalFiles = _totalFiles;
result.TotalDirectories = _totalDirectories;
result.TotalSize = _totalSize;
result.LargestFiles =
_largestFiles.OrderByDescending(f =>
f.Size).Take(TOP_FILES_COUNT).ToList();
        result.LargestDirectories =
_largestDirectories.OrderByDescending(d =>
d.TotalSize).Take(TOP_DIRECTORIES_COUNT).ToList();

        // Розрахунок відсотків для статистики типів
файлів
foreach (var stat in _fileTypeStatistics.Values)
{
    stat.CalculatePercentage(_totalSize);
}

        result.FileTypeStatistics =
_fileTypeStatistics.Values.OrderByDescending(s =>
s.TotalSize).ToList();

        // Фінальне оновлення прогресу
if (progress != null)
{
    progress.Report(new ScanProgress
    {
        CurrentPath = path,
        CurrentDirectory = rootDirectory,
        Percentage = 100,
        FilesScanned = _totalFiles,
        TotalSize = _totalSize
    });
}

```

```

        // Завершення сканування
        result.Complete();

        Debug.WriteLine($"ScanDirectory завершено:");
        Debug.WriteLine($"  Всього файлів:
{_totalFiles}");
        Debug.WriteLine($"  Всього каталогів:
{_totalDirectories}");
        Debug.WriteLine($"  Загальний розмір:
{_totalSize} байт");
        Debug.WriteLine($"  Найбільші файли:
{_largestFiles.Count}");
        Debug.WriteLine($"  Типи файлів:
{_fileTypeStatistics.Count}");

        return result;
    }
    catch (OperationCanceledException)
    {
        // Сканування було скасовано
        ScanResult result = new ScanResult(path);
        result.Cancel();
        throw;
    }
    catch (Exception ex)
    {
        // Помилка при скануванні
        ScanResult result = new ScanResult(path);
        result.SetError(ex.Message);
        return result;
    }
}

/// <summary>
/// Рекурсивно сканує каталог та його підкаталоги
/// </summary>
/// <param name="path">Шлях до каталогу</param>
/// <param name="parent">Батьківський каталог</param>
/// <param name="progress">Інтерфейс для звітування про
прогрес</param>
/// <param name="cancellation_token">Токен для
скасування</param>
/// <param name="totalDriveSize">Загальний розмір диска
для розрахунку прогресу</param>
/// <returns>Об'єкт DirectoryInfo з інформацією про
каталог</returns>
private DirectoryInfo ScanDirectoryRecursive(
    string path,
    DirectoryInfo? parent,
    IProgress<ScanProgress>? progress,
    CancellationToken cancellation_token,
    long totalDriveSize)

```

```

{
    // Перевірка скасування
    cancellationToken.ThrowIfCancellationRequested();

    // Створення об'єкта каталогу
    DirectoryInfo directory = new DirectoryInfo(path)
    {
        Parent = parent
    };

    try
    {
        // Отримання інформації про каталог
        DirectoryInfo dirInfo = new DirectoryInfo(path);

        // Сканування файлів у каталозі
        ScanFilesInDirectory(dirInfo, directory,
cancellationToken);

        // Сканування підкаталогів
        ScanSubdirectories(dirInfo, directory, progress,
cancellationToken, totalDriveSize);

        // Оновлення загального розміру каталогу
        directory.UpdateTotalSize();

        // Додавання до списку найбільших каталогів
        AddToLargestDirectories(directory);

        // Звіт про прогрес (кожні N каталогів)
        _directoriesSinceLastProgress++;
        if (_directoriesSinceLastProgress >=
DIRECTORIES_PER_PROGRESS_UPDATE)
        {
            ReportProgress(path, directory, progress,
totalDriveSize);
            _directoriesSinceLastProgress = 0;
        }

        _totalDirectories++;
    }
    catch (UnauthorizedAccessException)
    {
        // Пропускаємо каталоги без доступу
        Debug.WriteLine($"Немає доступу до каталогу:
{path}");
    }
    catch (DirectoryNotFoundException)
    {
        // Каталог не знайдено
        Debug.WriteLine($"Каталог не знайдено: {path}");
    }
}

```

```

        catch (IOException ex)
        {
            // Помилка вводу-виводу
            Debug.WriteLine($"Помилка вводу-виводу для
каталогу {path}: {ex.Message}");
        }
        catch (Exception ex)
        {
            // Інші помилки
            Debug.WriteLine($"Помилка при скануванні
каталогу {path}: {ex.Message}");
        }

        return directory;
    }

    /// <summary>
    /// Сканує файли у заданому каталозі
    /// </summary>
    /// <param name="dirInfo">Інформація про каталог</param>
    /// <param name="directory">Об'єкт каталогу для
заповнення</param>
    /// <param name="cancellationToken">Токен для
скасування</param>
    private void ScanFilesInDirectory(
        DirectoryInfo dirInfo,
        DirectoryInfo directory,
        CancellationToken cancellationToken)
    {
        try
        {
            // Отримання файлів у каталозі
            FileInfo[] files = dirInfo.GetFiles();

            // Логування тільки якщо є файли
            if (files.Length > 0 && _totalFiles % 1000 == 0)
            {
                Debug.WriteLine($"Сканування
{dirInfo.FullName}: {files.Length} файлів");
            }

            foreach (FileInfo file in files)
            {
                // Перевірка скасування
                cancellationToken.ThrowIfCancellationRequested();

                try
                {
                    // Створення об'єкта файлу
                    FileItem fileItem = new FileItem(
                        file.FullName,

```

```

        file.Length,
FileHelper.GetFileExtension(file.FullName),
        file.LastWriteTime);

// Додавання файлу до каталогу
directory.AddFile(fileItem);

// Оновлення статистики
_totalFiles++;
_totalSize += file.Length;

// Додавання до списку найбільших файлів
AddToLargestFiles(fileItem);

// Оновлення статистики по типах файлів
UpdateFileTypeStatistics(fileItem);
}
catch (UnauthorizedAccessException)
{
    // Пропускаємо файли без доступу
    Debug.WriteLine($"Немає доступу до
файлу: {file.FullName}");
}
catch (Exception ex)
{
    // Помилка при обробці файлу
    Debug.WriteLine($"Помилка при обробці
файлу {file.FullName}: {ex.Message}");
}
}
}
catch (UnauthorizedAccessException)
{
    // Пропускаємо каталоги без доступу
    Debug.WriteLine($"Немає доступу до файлів у
каталозі: {dirInfo.FullName}");
}
catch (Exception ex)
{
    // Інші помилки
    Debug.WriteLine($"Помилка при скануванні файлів
у каталозі {dirInfo.FullName}: {ex.Message}");
}
}

/// <summary>
/// Рекурсивно сканує підкаталоги
/// </summary>
/// <param name="dirInfo">Інформація про батьківський
каталог</param>

```

```

        /// <param name="directory">Об'єкт батьківського
каталогу</param>
        /// <param name="progress">Інтерфейс для звітування про
прогрес</param>
        /// <param name="cancellationToken">Токен для
скасування</param>
        /// <param name="totalDriveSize">Загальний розмір
диска</param>
        private void ScanSubdirectories(
            DirectoryInfo dirInfo,
            DirectoryInfo directory,
            IProgress<ScanProgress>? progress,
            CancellationToken cancellationToken,
            long totalDriveSize)
        {
            try
            {
                // Отримання підкаталогів
                DirectoryInfo[] subdirs =
dirInfo.GetDirectories();

                foreach (DirectoryInfo subdir in subdirs)
                {
                    // Перевірка скасування

cancellationToken.ThrowIfCancellationRequested();

                    try
                    {
                        // Рекурсивне сканування підкаталогу
                        DirectoryInfo subdirectory =
ScanDirectoryRecursive(
                            subdir.FullName,
                            directory,
                            progress,
                            cancellationToken,
                            totalDriveSize);

                        // Додавання підкаталогу до
батьківського
                        directory.AddSubdirectory(subdirectory);
                    }
                    catch (UnauthorizedAccessException)
                    {
                        // Пропускаємо підкаталоги без доступу
                        Debug.WriteLine($"Немає доступу до
підкаталогу: {subdir.FullName}");
                    }
                    catch (Exception ex)
                    {
                        // Інші помилки

```

```

        Debug.WriteLine($"Помилка при скануванні
підкаталогу {subdir.FullName}: {ex.Message}");
    }
}
catch (UnauthorizedAccessException)
{
    // Пропускаємо каталоги без доступу
    Debug.WriteLine($"Немає доступу до підкаталогів
у: {dirInfo.FullName}");
}
catch (Exception ex)
{
    // Інші помилки
    Debug.WriteLine($"Помилка при скануванні
підкаталогів у {dirInfo.FullName}: {ex.Message}");
}
}

/// <summary>
/// Додає файл до списку найбільших файлів
/// </summary>
/// <param name="file">Файл для додавання</param>
private void AddToLargestFiles(FileItem file)
{
    _largestFiles.Add(file);

    // Підтримуємо список обмеженого розміру
    if (_largestFiles.Count > TOP_FILES_COUNT * 2)
    {
        // Сортуємо та видаляємо найменші
        _largestFiles =
        _largestFiles.OrderByDescending(f =>
        f.Size).Take(TOP_FILES_COUNT * 2).ToList();
    }

    // Логування кожного 1000-го файлу
    if (_totalFiles % 1000 == 0)
    {
        Debug.WriteLine($"AddToLargestFiles: додано файл
{_totalFiles}, розмір списку: {_largestFiles.Count}");
    }
}

/// <summary>
/// Додає каталог до списку найбільших каталогів
/// </summary>
/// <param name="directory">Каталог для
додавання</param>
private void AddToLargestDirectories(DirectoryItem
directory)
{

```

```

        // Пропускаємо порожні каталоги
        if (directory.TotalSize == 0)
            return;

        _largestDirectories.Add(directory);

        // Підтримуємо список обмеженого розміру
        if (_largestDirectories.Count >
TOP_DIRECTORIES_COUNT * 2)
        {
            // Сортуємо та видаляємо найменші
            _largestDirectories =
            _largestDirectories.OrderByDescending(d =>
d.TotalSize).Take(TOP_DIRECTORIES_COUNT * 2).ToList();
        }
    }

    /// <summary>
    /// Оновлює статистику по типах файлів
    /// </summary>
    /// <param name="file">Файл для обліку в
статистиці</param>
    private void UpdateFileTypeStatistics(FileItem file)
    {
        string extension = file.Extension;

        // Якщо розширення пусте, використовуємо "[Без
розширення]"
        if (string.IsNullOrEmpty(extension))
        {
            extension = "[Без розширення]";
        }

        // Отримання або створення статистики для цього
розширення
        if (!_fileTypeStatistics.ContainsKey(extension))
        {
            _fileTypeStatistics[extension] = new
FileTypeStatistics(extension);
        }

        // Додавання файлу до статистики
        _fileTypeStatistics[extension].AddFile(file.Size);

        // Логування кожного 1000-го файлу
        if (_totalFiles % 1000 == 0)
        {
            Debug.WriteLine($"UpdateFileTypeStatistics:
типів файлів: {_fileTypeStatistics.Count}");
        }
    }
}

```



```

100    /// <summary>
    /// Звіт про прогрес сканування
    /// </summary>
    private void ReportProgress(
        string currentPath,
        DirectoryInfo currentDirectory,
        IProgress<ScanProgress>? progress,
        long totalDriveSize)
    {
        try
        {
            if (progress != null)
            {
                // Розрахунок відсотка на основі розміру
                double percentage = totalDriveSize > 0
                    ? (double)_totalSize / totalDriveSize *

                        : 0;

                // Обмеження відсотка до 100%
                percentage = Math.Min(percentage, 100);

                // Створення об'єкта прогресу
                ScanProgress scanProgress = new ScanProgress
                {
                    CurrentPath = currentPath,
                    CurrentDirectory = currentDirectory,
                    Percentage = percentage,
                    FilesScanned = _totalFiles,
                    TotalSize = _totalSize
                };

                // Звіт про прогрес
                progress.Report(scanProgress);
            }
        }
        catch (Exception ex)
        {
            Debug.WriteLine($"Помилка при звітуванні про
прогрес: {ex.Message}");
        }
    }

    /// <summary>
    /// Очищає результати попереднього сканування
    /// </summary>
    private void ClearResults()
    {
        _largestFiles.Clear();
        _largestDirectories.Clear();
        _fileTypeStatistics.Clear();
        _totalFiles = 0;
    }

```

```

        _totalDirectories = 0;
        _totalSize = 0;
        _directoriesSinceLastProgress = 0;

        Debug.WriteLine("ClearResults: очищено результати
попереднього сканування");
    }
}
}

```

AnalysisService.cs

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using DiskAnalyzer.Models;
using DiskAnalyzer.Utills;

namespace DiskAnalyzer.Services
{
    /// <summary>
    /// Сервіс для аналізу даних сканування
    /// Виконує пошук дублікатів, аналіз типів файлів та іншу
    статистику
    /// </summary>
    public class AnalysisService
    {
        // Мінімальний розмір файлу для пошуку дублікатів (в
байтах)
        private const long MIN_FILE_SIZE_FOR_DUPLICATES = 1024;
// 1 KB

        /// <summary>
        /// Конструктор сервісу аналізу
        /// </summary>
        public AnalysisService()
        {
        }

        /// <summary>
        /// Знаходить дублікати файлів у результаті сканування
        /// Використовує двоетапний алгоритм: групування за
розміром, потім за хешем
        /// </summary>
        /// <param name="scanResult">Результат
сканування</param>
        /// <param name="progress">Інтерфейс для звітування про
прогрес</param>

```

```

        /// <param name="cancellationToken">Токен для
скасування</param>
        /// <returns>Список груп дублікатів</returns>
        public List<DuplicateGroup> FindDuplicates(
            ScanResult scanResult,
            IProgress<AnalysisProgress>? progress = null,
            CancellationToken cancellationToken = default)
        {
            try
            {
                if (scanResult.RootDirectory == null)
                    return new List<DuplicateGroup>();

                // Збираємо всі файли з результату сканування
                List<FileItem> allFiles =
CollectAllFiles(scanResult.RootDirectory);

                // Фільтруємо файли за мінімальним розміром
                List<FileItem> filesToCheck = allFiles
                    .Where(f => f.Size >=
MIN_FILE_SIZE_FOR_DUPLICATES)
                    .ToList();

                // Звіт про прогрес
                progress?.Report(new AnalysisProgress
                {
                    Stage = "Групування за розміром",
                    CurrentFile = "",
                    ProcessedFiles = 0,
                    TotalFiles = filesToCheck.Count,
                    Percentage = 0
                });

                // Етап 1: Групування файлів за розміром
                Dictionary<long, List<FileItem>> filesBySize =
GroupFilesBySize(filesToCheck, cancellationToken);

                // Етап 2: Групування файлів за хешем (тільки
для груп з більше ніж 1 файлом)
                List<DuplicateGroup> duplicateGroups =
FindHashDuplicates(
                    filesBySize,
                    progress,
                    cancellationToken);

                return duplicateGroups;
            }
            catch (OperationCanceledException)
            {
                throw;
            }
            catch (Exception ex)

```

```

        {
            throw new Exception($"Помилка при пошуку
дублікатів: {ex.Message}", ex);
        }
    }

    /// <summary>
    /// Асинхронно знаходить дублікати файлів
    /// </summary>
    /// <param name="scanResult">Результат
сканування</param>
    /// <param name="progress">Інтерфейс для звітування про
прогрес</param>
    /// <param name="cancellationToken">Токен для
скасування</param>
    /// <returns>Список груп дублікатів</returns>
    public async Task<List<DuplicateGroup>>
FindDuplicatesAsync(
    ScanResult scanResult,
    IProgress<AnalysisProgress>? progress = null,
    CancellationToken cancellationToken = default)
    {
        return await Task.Run(() =>
FindDuplicates(scanResult, progress, cancellationToken),
cancellationToken);
    }

    /// <summary>
    /// Збирає всі файли з каталогу рекурсивно
    /// </summary>
    /// <param name="directory">Кореневий каталог</param>
    /// <returns>Список всіх файлів</returns>
    private List<FileItem> CollectAllFiles(DirectoryItem
directory)
    {
        List<FileItem> allFiles = new List<FileItem>();

        // Додаємо файли з поточного каталогу
        allFiles.AddRange(directory.Files);

        // Рекурсивно додаємо файли з підкаталогів
        foreach (var subdir in directory.Subdirectories)
        {
            allFiles.AddRange(CollectAllFiles(subdir));
        }

        return allFiles;
    }

    /// <summary>
    /// Групує файли за розміром
    /// Файли з однаковим розміром можуть бути дублікатами

```

```

        /// </summary>
        /// <param name="files">Список файлів для
групування</param>
        /// <param name="cancellationToken">Токен для
скасування</param>
        /// <returns>Словник: розмір -> список файлів</returns>
        private Dictionary<long, List<FileItem>>
GroupFilesBySize(
            List<FileItem> files,
            Cancellation_token cancellationToken)
        {
            Dictionary<long, List<FileItem>> filesBySize = new
Dictionary<long, List<FileItem>>();

            foreach (var file in files)
            {
                cancellationToken.ThrowIfCancellationRequested();

                if (!filesBySize.ContainsKey(file.Size))
                {
                    filesBySize[file.Size] = new
List<FileItem>();
                }

                filesBySize[file.Size].Add(file);
            }

            // Видаляємо групи з одним файлом (не можуть бути
дублікатами)
            var groupsWithMultipleFiles = filesBySize
                .Where(kvp => kvp.Value.Count > 1)
                .ToDictionary(kvp => kvp.Key, kvp => kvp.Value);

            return groupsWithMultipleFiles;
        }

        /// <summary>
        /// Знаходить дублікати за хешем файлів
        /// Обчислює SHA256 хеш для кожної групи файлів з
однаковим розміром
        /// </summary>
        /// <param name="filesBySize">Групи файлів за
розміром</param>
        /// <param name="progress">Інтерфейс для звітування про
прогрес</param>
        /// <param name="cancellationToken">Токен для
скасування</param>
        /// <returns>Список груп дублікатів</returns>
        private List<DuplicateGroup> FindHashDuplicates(
            Dictionary<long, List<FileItem>> filesBySize,
            IProgress<AnalysisProgress>? progress,

```

```

        CancellationToken cancellationToken)
    {
        List<DuplicateGroup> duplicateGroups = new
List<DuplicateGroup>();

        int totalGroups = filesBySize.Count;
        int processedGroups = 0;

        foreach (var kvp in filesBySize)
        {

cancellationToken.ThrowIfCancellationRequested();

            long size = kvp.Key;
            List<FileItem> files = kvp.Value;

            // Звіт про прогрес
            processedGroups++;
            double percentage = (double)processedGroups /
totalGroups * 100;

            progress?.Report(new AnalysisProgress
            {
                Stage = "Обчислення хешів",
                CurrentFile = files.FirstOrDefault()?.Name
?? "",

                ProcessedFiles = processedGroups,
                TotalFiles = totalGroups,
                Percentage = percentage
            });

            // Групування файлів за хешем
            Dictionary<string, List<FileItem>> filesByHash =
new Dictionary<string, List<FileItem>>();

            foreach (var file in files)
            {

cancellationToken.ThrowIfCancellationRequested();

                try
                {
                    // Обчислення хешу файлу
                    string hash =
FileHelper.CalculateFileHash(file.Path);
                    file.Hash = hash;

                    // Додавання до групи за хешем
                    if (!filesByHash.ContainsKey(hash))
                    {
                        filesByHash[hash] = new
List<FileItem>();

```

```

        }

        filesByHash[hash].Add(file);
    }
    catch (Exception ex)
    {
        // Помилка при обчисленні хешу -
пропускаємо файл
        Debug.WriteLine($"Помилка обчислення
хешу для {file.Path}: {ex.Message}");
    }
}

// Створення груп дублікатів для груп з більше
ніж 1 файлом
foreach (var hashGroup in filesByHash)
{
    if (hashGroup.Value.Count > 1)
    {
        DuplicateGroup group = new
DuplicateGroup(hashGroup.Key, size);

        foreach (var file in hashGroup.Value)
        {
            group.AddDuplicate(file);
        }

        duplicateGroups.Add(group);
    }
}

// Сортювання груп за розміром зайвого простору (від
найбільшого)
return duplicateGroups
    .OrderByDescending(g => g.WastedSpace)
    .ToList();
}

/// <summary>
/// Аналізує розподіл файлів за типами (розширеннями)
/// </summary>
/// <param name="scanResult">Результат
сканування</param>
/// <returns>Список статистики по типах файлів</returns>
public List<FileTypeStatistics>
AnalyzeFileTypes(ScanResult scanResult)
{
    try
    {
        if (scanResult.RootDirectory == null)
            return new List<FileTypeStatistics>();
    }
}

```

```

        // Використовуємо вже зібрану статистику з
результату сканування
        // Якщо статистика ще не зібрана, збираємо її
        if (scanResult.FileTypeStatistics == null ||
scanResult.FileTypeStatistics.Count == 0)
        {
            scanResult.FileTypeStatistics =
CollectFileTypeStatistics(scanResult.RootDirectory);
        }

        // Обчислення відсотків
        foreach (var stat in
scanResult.FileTypeStatistics)
        {

stat.CalculatePercentage(scanResult.TotalSize);
        }

        return scanResult.FileTypeStatistics;
    }
    catch (Exception ex)
    {
        throw new Exception($"Помилка при аналізі типів
файлів: {ex.Message}", ex);
    }
}

/// <summary>
/// Збирає статистику по типах файлів з каталогу
/// </summary>
/// <param name="directory">Каталог для аналізу</param>
/// <returns>Список статистики по типах файлів</returns>
private List<FileTypeStatistics>
CollectFileTypeStatistics(DirectoryItem directory)
{
    Dictionary<string, FileTypeStatistics> statistics =
new Dictionary<string, FileTypeStatistics>();

    CollectFileTypeStatisticsRecursive(directory,
statistics);

    return statistics.Values.ToList();
}

/// <summary>
/// Рекурсивно збирає статистику по типах файлів
/// </summary>
/// <param name="directory">Каталог</param>
/// <param name="statistics">Словник статистики</param>
private void CollectFileTypeStatisticsRecursive(
    DirectoryItem directory,

```



```

Dictionary<string, FileTypeStatistics> statistics)
{
    // Обробка файлів у поточному каталозі
    foreach (var file in directory.Files)
    {
        string extension = file.Extension;

        if (string.IsNullOrEmpty(extension))
        {
            extension = "[Без розширення]";
        }

        if (!statistics.ContainsKey(extension))
        {
            statistics[extension] = new
FileTypeStatistics(extension);
        }

        statistics[extension].AddFile(file.Size);
    }

    // Рекурсивна обробка підкаталогів
    foreach (var subdir in directory.Subdirectories)
    {
        CollectFileTypeStatisticsRecursive(subdir,
statistics);
    }
}

/// <summary>
/// Знаходить найбільші файли в результаті сканування
/// </summary>
/// <param name="scanResult">Результат
сканування</param>
/// <param name="count">Кількість файлів для
повернення</param>
/// <returns>Список найбільших файлів</returns>
public List<FileItem> FindLargestFiles(ScanResult
scanResult, int count = 100)
{
    try
    {
        if (scanResult.RootDirectory == null)
            return new List<FileItem>();

        // Збираємо всі файли
        List<FileItem> allFiles =
CollectAllFiles(scanResult.RootDirectory);

        // Сортуюмо за розміром та повертаємо TOP-N
        return allFiles
            .OrderByDescending(f => f.Size)

```

```

        .Take(count)
        .ToList();
    }
    catch (Exception ex)
    {
        throw new Exception($"Помилка при пошуку
найбільших файлів: {ex.Message}", ex);
    }
}

/// <summary>
/// Знаходить найбільші каталоги в результаті сканування
/// </summary>
/// <param name="scanResult">Результат
сканування</param>
/// <param name="count">Кількість каталогів для
повернення</param>
/// <returns>Список найбільших каталогів</returns>
public List<DirectoryItem>
FindLargestDirectories(ScanResult scanResult, int count = 50)
{
    try
    {
        if (scanResult.RootDirectory == null)
            return new List<DirectoryItem>();

        // Збираємо всі каталоги
        List<DirectoryItem> allDirectories =
CollectAllDirectories(scanResult.RootDirectory);

        // Сортуюмо за розміром та повертаємо TOP-N
        return allDirectories
            .OrderByDescending(d => d.TotalSize)
            .Take(count)
            .ToList();
    }
    catch (Exception ex)
    {
        throw new Exception($"Помилка при пошуку
найбільших каталогів: {ex.Message}", ex);
    }
}

/// <summary>
/// Збирає всі каталоги рекурсивно
/// </summary>
/// <param name="directory">Кореневий каталог</param>
/// <returns>Список всіх каталогів</returns>
private List<DirectoryItem>
CollectAllDirectories(DirectoryItem directory)
{

```

```

        List<DirectoryItem> allDirectories = new
List<DirectoryItem>();

        // Додаємо поточний каталог
        allDirectories.Add(directory);

        // Рекурсивно додаємо підкаталоги
        foreach (var subdir in directory.Subdirectories)
        {

allDirectories.AddRange(CollectAllDirectories(subdir));
        }

        return allDirectories;
    }
}

/// <summary>
/// Клас для передачі інформації про прогрес аналізу
/// </summary>
public class AnalysisProgress
{
    /// <summary>
    /// Поточний етап аналізу
    /// </summary>
    public string Stage { get; set; } = string.Empty;

    /// <summary>
    /// Ім'я поточного файлу, що обробляється
    /// </summary>
    public string CurrentFile { get; set; } = string.Empty;

    /// <summary>
    /// Кількість оброблених файлів
    /// </summary>
    public int ProcessedFiles { get; set; }

    /// <summary>
    /// Загальна кількість файлів для обробки
    /// </summary>
    public int TotalFiles { get; set; }

    /// <summary>
    /// Відсоток завершення
    /// </summary>
    public double Percentage { get; set; }
}
}

```